An object oriented approach to electrical machine design

Norton, Mark B.

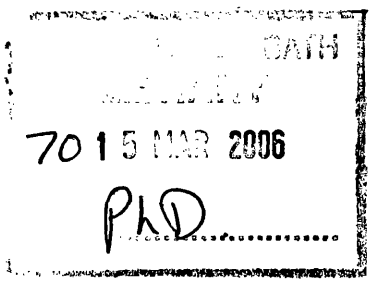
Award date:
2005

Awarding institution:
University of Bath

AN OBJECT ORIENTED APPROACH TO ELECTRICAL MACHINE DESIGN

Submitted by M. B. Norton
for the degree of
Doctor of Philosophy
of the University of Bath
2005

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346


BATH
7015 MAR 2006
PhD

Acknowledgements

Thank you to my supervisor Dr P.J. Leonard, for his patience, help and encouragement over the past years. This work would be nowhere near complete without his support.

Dr P.J. Coles deserves special mention for his contribution of the high speed motor and guidance in its modelling, an excellent real world example of an electrical machine which has been used within this work.

I would like to gratefully acknowledge the financial support of the Engineering and Physical Sciences Research Council (EPSRC).

Thanks to Bojan Niceno and Jonathan Shewchuk for making their meshing programs Easymesh[1] and Triangle[2] freely available to researchers in need of additional meshing algorithms.

Thanks to my parents and Sue for all their support, especially for the loan of the printing requisites.

The typesetting language LaTeX deserves mentioning for its help in alleviating the presentation headaches, allowing me to concentrate on the content.

Summary

Modelling of the electrical machine is taken from an object oriented design approach. Groups of objects have been identified that allow any electrical machine to be modelled. Objects range from values that parameterise any aspect of the design through to graphical primitives that construct the machine's geometry. More complex objects utilise the symmetry within the machine to reduce the amount of modelling involved by replicating geometries and machine properties, allowing transformations during the mapping process. The end goal is to construct a whole machine from the smallest component of symmetry, geometrically copying it to produce the entire machine whilst allowing transformations to handle varying region and boundary properties. These components can be saved and imported into new models, parameterisation of the entire component allows them to be slotted into place in any new design.

A simple language is used to represent these objects and thus the electrical machine. The language describes objects in terms of each other so that the state of the model can be stored in a dependency tree. Changes to the parameterisation result in manipulation of the dependency tree and these changes filter down the tree with immediate effect. This allows the entire machine geometry to change shape as a slot tooth is widened, it allows refinement of the mesh in response to numerical changes, and it also allows reconfiguration of an entire stator winding even though we only designed a single slot. Many objects exist within the groups identified and these can all be parameterised and interchanged, programatically we can make unlimited additions to these groups without risk of breaking the program. Manipulation of the dependency tree allows objects to be pulled out of the tree and substituted with a replacement, this implements the changes made to the model such that every change can be undone and redone with a history that goes back as far as required. Manipulating the dependency tree means that

none of the objects need knowledge of how to undo themselves, we have a generic undo/redo mechanism.

The designer is now free to interchange objects, substitute one meshing algorithm for another, swap between library components of different machine parts, always with the ability to undo and redo changes and achieve greater freedom in the design process.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

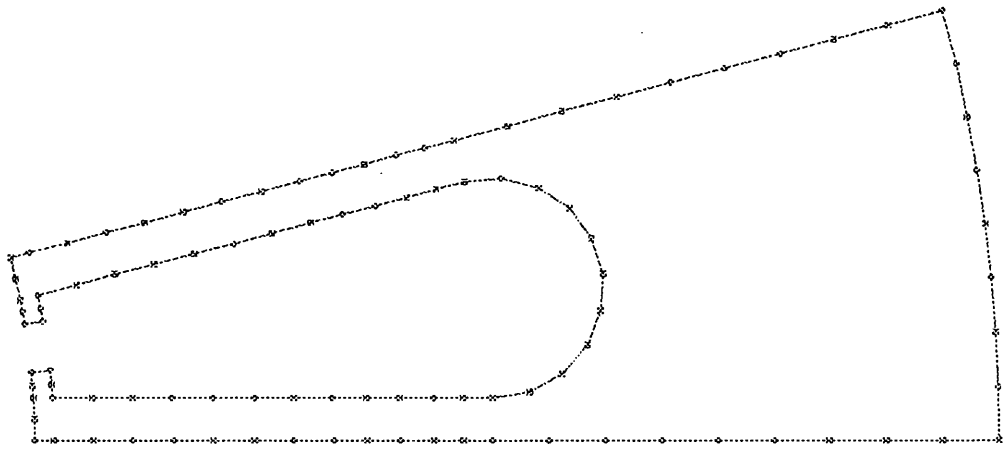


Figure 1.1: The Beginnings Of An Electrical Machine Model, Nodal Outline Of A Stator Slot

Figure 1.1 outlines the beginnings of an electrical machine design. Whilst the design process may be more involved than this simplified figure details the foundation of a mesh can be seen, a meshing algorithm may be all that is needed to complete the layout of nodes and interconnecting elements as seen in the mesh of figure 1.2.

1.1 Previous Work

Much work has been done on the generation of a mesh given the initial outlining geometry, such as the geometry of figure 1.2.

However very little work has been done, and even less published, on programs specific to defining the initial shape or geometry ready for finite element meshing. This is probably because such schemes are implemented in commercial programs and the methods have not been published. Programs such as SPEED[3], from the University of Glasgow, and JMAG-Studio[4], from The Japan Research Institute, hint to this.

SPEED generates the coordinates of nodes along all region boundaries, a process referred to as Unimesh. The meshing of these areas is then performed as a separate process within the software.

JMAG-Studio has the capability to read CAD data, such as DXF files generated by Autocad[5], so that "Analysis is implemented using the same shape data that is used in design" which suggests that the design process is initiated in the CAD package.

A CAD based approach allows the geometry to be defined in terms of graphical primitives, such as lines, arcs, circles, and splines. Once the outline of the mesh has been drawn it can be handed to an unstructured meshing algorithm to produce the final mesh. However generic CAD based packages are not specifically aimed at meshing and so, for example, cannot specify the node density or any other constraints on the mesh. Neither are they specifically aimed at the design of electrical machines and so, for instance, they do not allow the utilisation of symmetry within an electrical machine which would allow only a single rotor segment to be drawn.

A dedicated design process is needed for electrical machines that can take elements of the CAD process and produce the outline of the electrical machine's geometry. This process can then benefit from the wealth of meshing algorithms available and use such algorithms to complete the finite element mesh outlined by this geometry. Once changes to the geometry are made, through parameterisation, the finite element mesh is reconstructed.

There are some publications in this area. A generic extensible geometry interface[6] focuses on the interchange of geometric data between solid modelling programs, translating different representations into its own. Related heavily to the use of CAD systems, this work allows the interchange of data between several well known solid modellers. It highlights that for a given geometry the mesh often needs to be tailored for different solutions and that producing the geometry in one program whilst generating the mesh in another is problematic. The interface allows its own geometric representation, translated from others, to be read in terms of points, curves, and surfaces.

On a similar theme the object-oriented virtual geometry interface[7] focuses not on one geometric representation but on the ability to read one representation and translate to several, translating to the best representation for a given problem. Solid model, faceted, composite, and mesh based representations are briefly discussed but the work focuses on object oriented techniques that allow access and translation between the geometric representations, whilst allowing t of other representations.

Work has been done using parameterised templates[8] where the outlining geometry is still defined using nodes, however labels are given to these nodes so they can be identified easily. This process adds a layer of abstraction. Before a meshing algorithm is used to mesh the enclosed area a process locates nodes by their label and updates their coordinates.

The designer benefits from being able to reuse past mesh templates in new designs and the technique is geared towards optimisation, demonstrating the advantages of automated mesh refinement. However there is no relationship between the labelled nodes. Refining the mesh is not as simple as updating a single parameter, such as the airgap width, each refinement of the mesh involves moving a group of nodes.

An object oriented approach to mesh refinement[9] is an example of the wealth of papers that demonstrate the advantages of object oriented design, such as the ability to easily maintain, extend, and understand the program. This work focuses on the refinement of the finite element mesh without changing the defining geometry.

More relevant work[10], again in the field of mesh refinement, deals with the use object oriented techniques used to describe domains within the whole geometry, termed the manifold. Geometric objects describe vertices, triangles, and line segments, the line segments being used to describe partitions within the manifold so that they may be meshed independently. Partitions are chosen to define smaller, more structured, meshes which are like tiles that combined together form a single mesh called the composite mesh.

Finally, a search of the internet highlights an internal report that outlines an object oriented finite element meshing system[11] which is the most related work to be found. In this method extensive templating has been incorporated to separate the containing geometry from the mesh algorithm, with application to stiffness matrix generation. This work defines a surface which is a two-dimensional arbitrary region whose extremities are defined by a number of curves, these being abstract objects that can equate to lines and arcs. Each of these curves is defined by a number of points and additional points are added to the curves during the meshing of the area.

1.2 This Work

This thesis explores the use of object oriented methods that generate the outlining geometry of an electrical machine and its finite element mesh, specifically for two-dimensional problems.

A tool is provided that allows the designer to produce a parameterised geometric description of the electrical machine so that parameters such as the airgap width, slot depth, and node density can be varied and the new mesh generated in one single action.

This is achieved by defining a set of objects that construct the electrical machine in a bottom-up design. A relationship is established between these objects that allows the variation of one parameter to cascade throughout all affected objects so that an updated mesh is produced.

In turn the advantages of this object oriented method allow the defining objects

of the machine to be interchanged at will so as to influence or change the design. A process that records such changes and allows the designer to step backwards and forwards, undoing and redoing the changes that were made.

At the top-level of this design process sits a component that can describe a symmetrical part of the electrical machine, utilising that symmetry to copy slots and windings and construct the entire machine whilst simultaneously handling variations in phase and polarity of windings and other regions.

These components can be stored as a library of interchangeable parts allowing reuse and modification into future designs.

The aim has been to write a program for the design of electrical machines that requires the least amount of effort from the designer.

Although this thesis uses a rotating machine as an example the methodology is general and could equally well be used for a wide variety of applications.

1.3 Designing An Electrical Machine With The Least Amount Of Effort

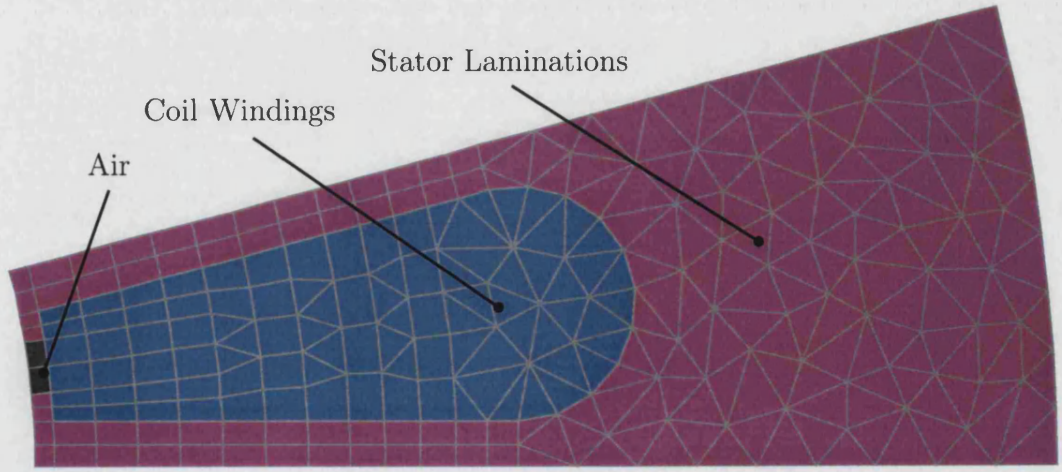


Figure 1.2: The Beginnings Of An Electrical Machine Model, Stator Slot Meshed And Elements Colour Coded According To Different Regions Of Material

The design process can often exploit the symmetry within an electrical machine so that the slot of figure 1.2 can be copied and rotated through an angle of *a2* degrees as seen in figure 1.3. Provided the nodes correspond along the joined edge of the two slot sections we can copy slots until the entire Stator is formed, saving a considerable amount of effort on the part of the designer.

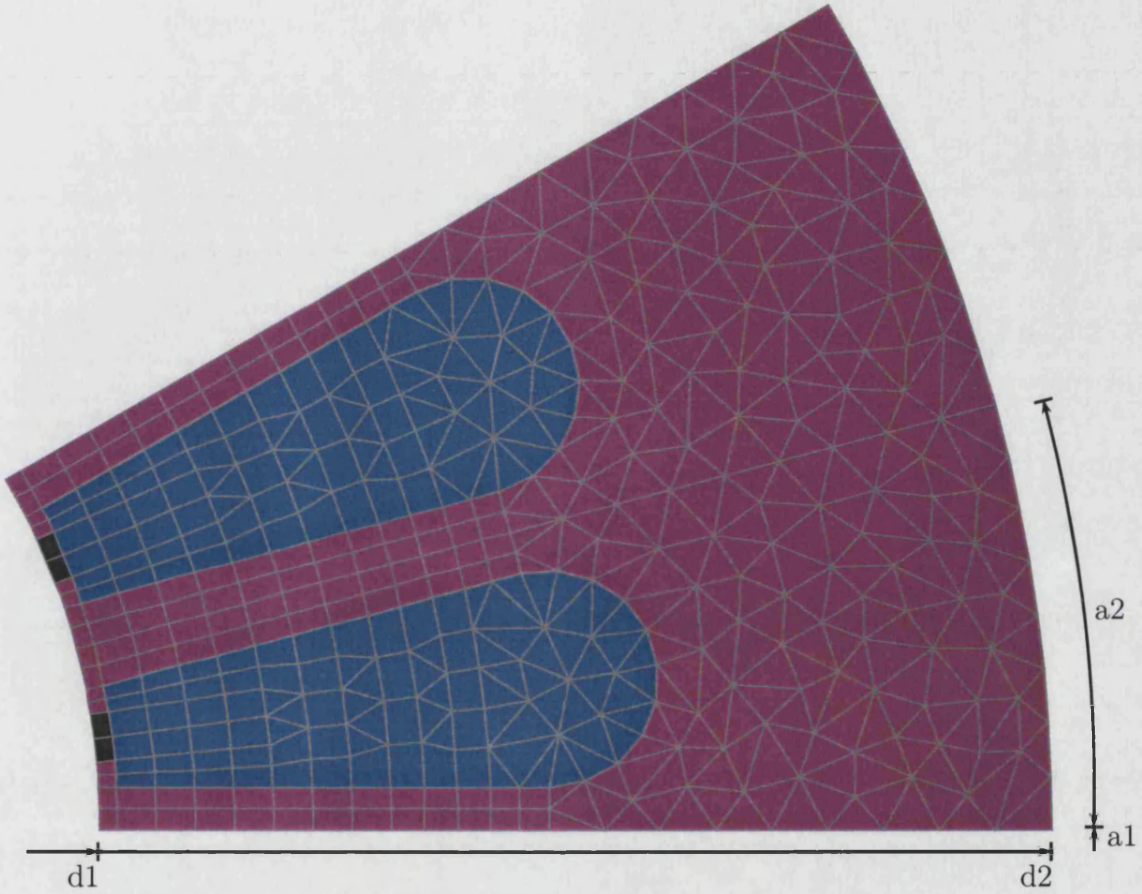


Figure 1.3: Reproducing The Slot To Exploit A Machine's Symmetry And Reduce The Amount Of Manual Meshing Involved

With modification of coil regions within the slot, to account for the differing polarity and phase of conductors carried in each slot, we will have achieved the complete stator mesh of figure 1.4. This mesh can be represented as a list of nodes and a list of elements, each element providing the node connectivity and a means of identifying the material properties of the region in which its included. Its a straight forward means of representation so the tools needed to build this mesh can vary considerably in complexity depending on how much work they take away from the designer. It's when modification of the mesh is required that this is fully appreciated.

Figure 1.4: Final Stator Mesh

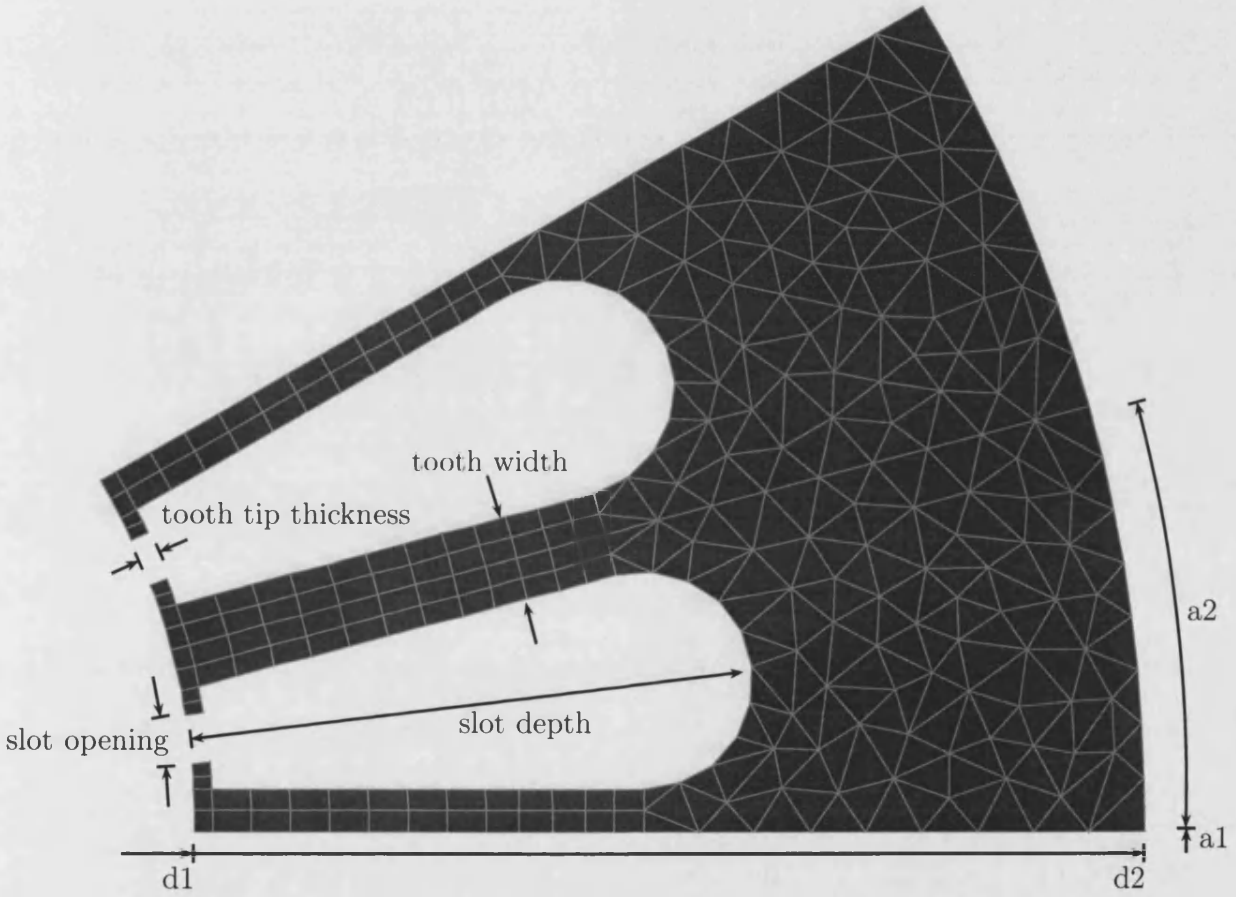


Figure 1.5: Parameterisation Of Stator Slot

If we're to modify our mesh to change the width of the slot opening we can probably achieve this without much effort, modify the slot depth and we could be looking at much more work. Given the boundary of nodes of figure 1.1 it would be possible to redefine the boundary and resubmit it to some meshing algorithm, freely available programs such as Bojan Niceno's Easymesh[1] and Jonathan Shewchuk's Triangle[2] perform such a task. We'd need to exploit the machine's symmetry again and copy the slot through an angular rotation in order to rebuild the entire machine, then there's still the problem of defining the numerous regions that can be seen in figure 1.4. In our example each slot carries two layers of conductors, with 24 slots we're potentially redefining 48 regions of material. Should we fundamentally change the geometry, or decide to experiment with different stator winding configurations, we're looking at considerable work in redefining these regions.

1.4 Using Parameterisation In Our Design

This is where parameterisation helps us by reducing the amount of work required to modify the mesh. There's no single way to parameterise a model, methods exist that allow the mesh to be reformed but we're looking to parameterise the geometry of figure 1.1. Figure 1.5 shows the geometric parameters we'll use to modify our boundary of nodes, we can then mesh this area using any meshing algorithm we desire allowing further parameterisation of the input of the meshing algorithm so we can, for instance, define values that restrain the minimum angles and maximum areas used for elements in a mesh.

To fine tune the mesh density we can also allow control over the number of nodes used along each edge of our boundary. The figures seen so far demonstrate the use of two meshing algorithms, used with our parametric design, the slot tooth has been meshed using an alternative algorithm to the outer area of the stator which has been meshed using Triangle[2]. The ability to allow interchangeable meshing algorithms within the design is possible because we can define the mesh as one entity or break it down into smaller parts called *mesh tiles.* Using an object oriented approach we will demonstrate how the whole electrical machine can be broken down into a series of parts, represented by objects in our object oriented scheme, that can be fitted together in a flexible manner and interchanged

to suit the designer. To fine tune meshing we can further divide our areas into smaller mesh tiles which can be swapped for other mesh tiles that use different meshing algorithms with different parameter values restraining angles and areas of elements.

1.5 Objects Used For The Electrical Machine

Figure 1.5 shows all geometric parameters we have chosen in order to modify this particular design of slot. The most important parameters here are *d1*, *d2*, *a1* and *a2* which define the extremities of our slot. If we describe any slot using these parameters then it becomes an interchangeable *component*, as we shall come to describe it, of an electrical machine which can be stored amongst a library of components and imported for reuse in future designs.

The further parameterisations of figure 1.5, *tooth tip thickness*, *tooth width*, *slot opening* and *slot depth*, are dependent on the slot design and not essential for our parameterisation. These parameters supplement our *d1*, *d2*, *a1*, *a2* parameterisation and merely increase the flexibility of the design, we shall focus on the primary *d1*, *d2*, *a1* and *a2* parameterisation to explain how the design works.

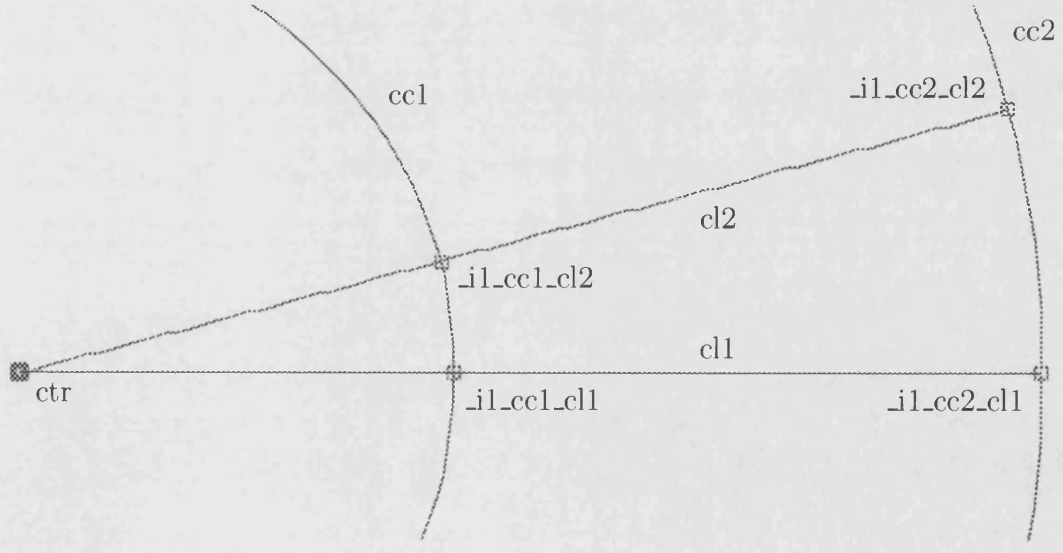


Figure 1.6: Parameterising The Stator Slot As A Single Part Of An Electrical Machine

Consider figure 1.6 where we have defined a centre point, *ctr*, and from this drawn

two lines and two circles using the point as their origin. The following language is used to describe this construction:

```
d1 = 27.75
d2 = 67.5
a1 = 0
a2 = 15
ctr = vvpoint(0, 0)

cc1 = pvcircle(ctr, d1)
cc2 = pvcircle(ctr, d2)
cl1 = pvvline(ctr, d2, a1)
cl2 = pvvline(ctr, d2, a1 + a2)
```

Here we have first defined the values of parameterisation, *d1*, *d2*, *a1*, and *a2*, followed by the point *ctr* which is also a key parameter in our definition. Next the circle *cc1* is constructed with *ctr* as its origin and *d1* as its diameter, *cc2* is constructed similarly and then two lines are defined with a point of origin, length, and angle.

The above description is used to save the state of this geometry, the entire machine is described in this manner rather than a node, element representation. Each line in this description results in the building of a value, point, or line segment object and these objects are stored in a dependency tree. Figure 1.7 shows the relationship of objects in this dependency tree.

What we haven't defined in the description of this machine are the intersection points *_i1_cc1_cl1*, *_i1_cc2_cl1*, *_i1_cc1_cl3*, and *_i1_cc3_cl3* illustrated in figure 1.6. These points exist where the line segments intersect, *_i1_cc1_cl1* being the intersection of circle *cc1* and line *cl1*, and have been automatically added by a post-processing routine within our program.

These points are useful because we can take any two points along a line segment and use them to define a line of nodes, called a *discrete segment.* Figure 1.8 shows four discrete segments, each described by one of the four line segments and its intersection with the other line segments. The result is an area completely bounded by nodes, defined by the following syntax:

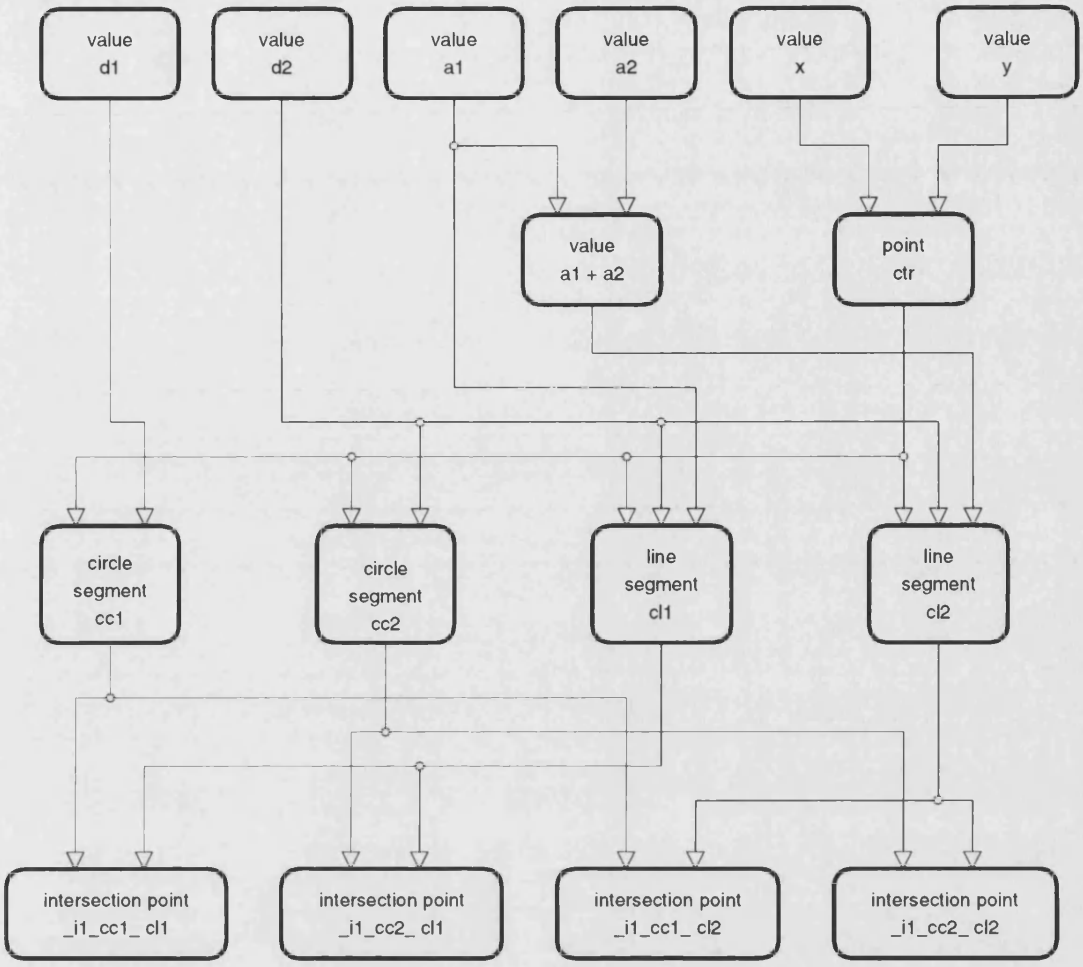


Figure 1.7: Corresponding Dependency Tree For The Value, Point, And Segment Objects Of Figure 1.6

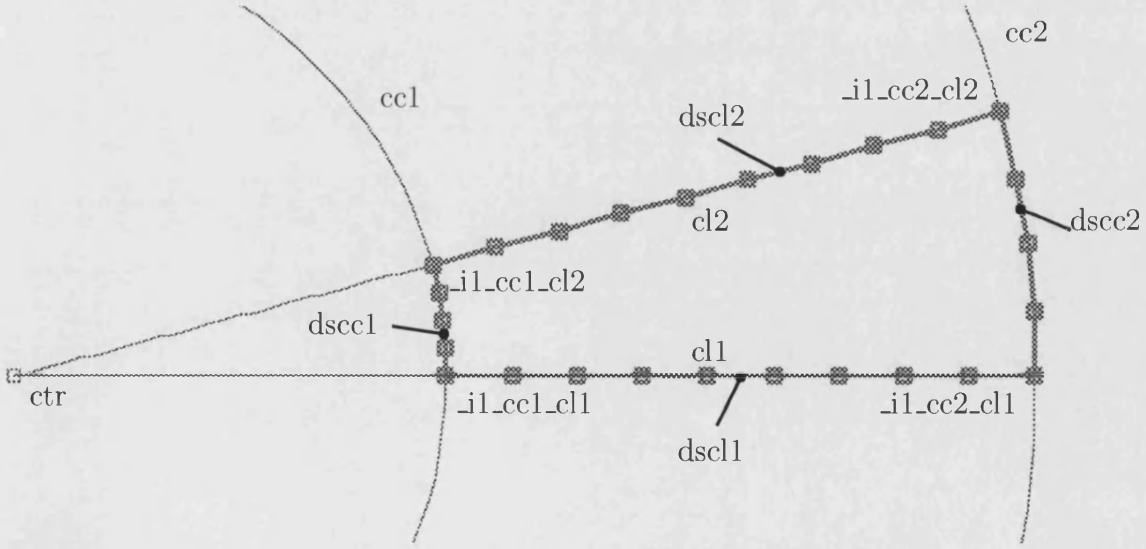


Figure 1.8: Using A Segment Bounded By Two Points To Form A Line Of Nodes

```
n = 1.0

dscl1 = dsegment(cl1, _i1_cc1_cl1, _i1_cc2_cl1, 10n)
dscl2 = dsegment(cl2, _i1_cc1_cl2, _i1_cc2_cl2, 10n)
dscc1 = dsegment(cc1, _i1_cc1_cl1, _i1_cc1_cl2, 5n)
dscc2 = dsegment(cc2, _i1_cc2_cl1, _i1_cc2_cl2, 5n)
```

To create a discrete segment we provide it with the name of a segment and two points which must lie along the segment's path. In this example we've used the intersection points which are automatically created and updated by our program. A final argument defines the number of nodes that are to populate the length of the segment. Here we have chosen to parameterise this number by making it a multiple of n, we can now vary the density of a mesh bounded by this area by adjusting this parameter. To mesh this area we simply provide these discrete segments as the arguments to one of the available meshing algorithms:

```
meshtile1 = semesh(dscl1, dscl2, dscc1, dscc2)
```

This produces the mesh of figure 1.9. The following figure, 1.10, shows the density of the mesh increase proportionally to the increase of n, the value we chose to parameterise our mesh density. This is simply done by assigning a new value to the parameter:

```
n = 1.5
```

Figure 1.11 then shows complete replacement of the mesh with a mesh tile that uses the Triangle[2] meshing program, again through reassignment of an existing variable. With this type of mesh tile we can control the mesh density not only through the discrete segments but also through two numerical values, the first restraining the minimum angle used in the mesh and the latter restraining the maximum area for any one element.

```
meshtile1 = triangle(dscl1, dscl2, dscc1, dscc2, 0, 20)
```

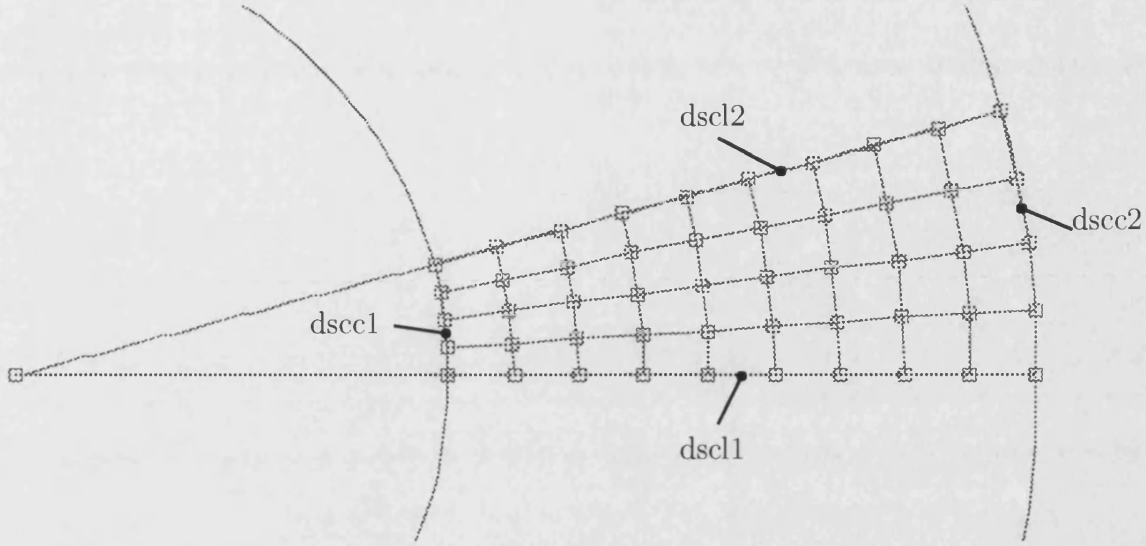


Figure 1.9: Mesh Tile Defined By Discrete Segments

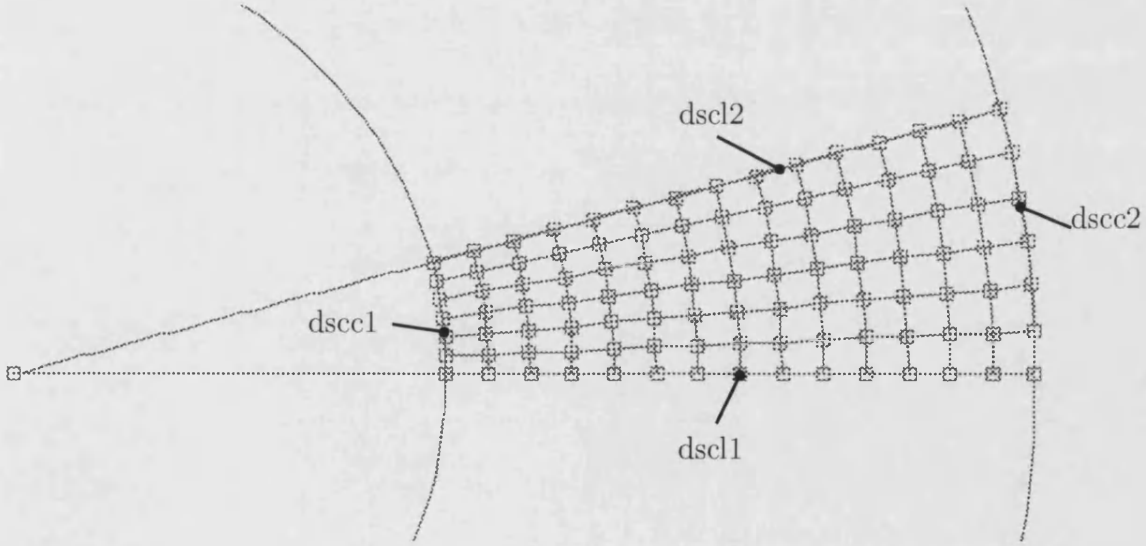


Figure 1.10: Mesh Density Increased Through Chosen Parameterisation Value n

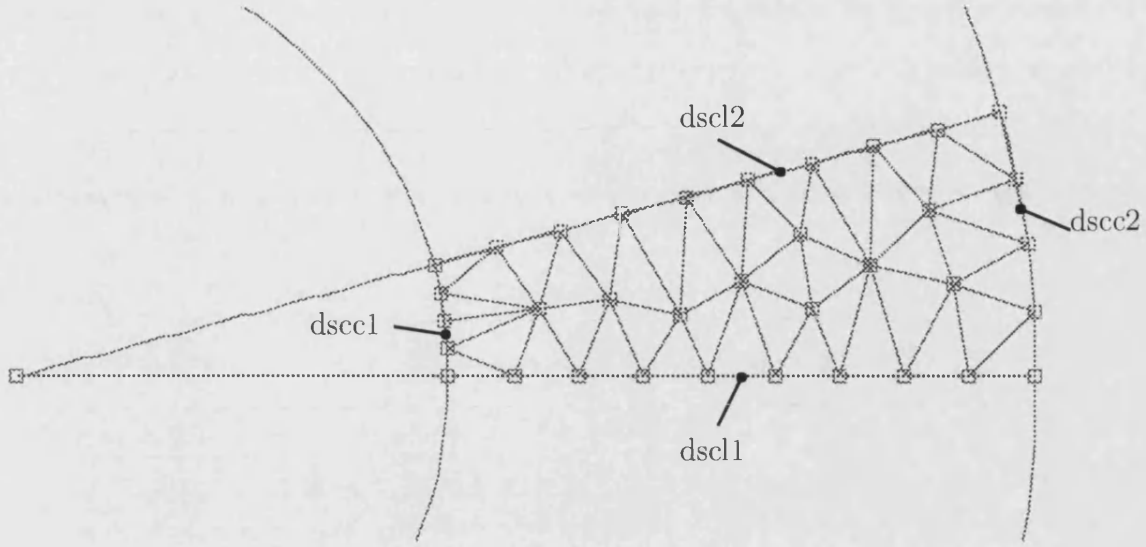


Figure 1.11: Incorporating An Alternative Meshing Algorithm, Jonathan Shewchuk's Triangle[2]

1.6 An Inherent Undo/Redo Mechanism

When it comes to the reassignment of objects, such as the mesh tiles illustrated above, we are actually removing the old mesh tile from the dependency tree and replacing it with the new mesh tile. The old mesh tile isn't discarded, it's kept in an undo buffer so at any point we can reverse this change, the old mesh tile is reinserted into the dependency tree and its replacement is removed and placed in the redo buffer.

The implications of this are that mesh tiles don't have to know how to undo themselves, they are simply replaced with another mesh tile. This works for any of our objects devised in this object oriented approach whether they be values, points, line segments or mesh tiles. An undo/redo mechanism simply operates on replacement of an object with another object of the same kind, such as a line segment and arc segment or two mesh tiles with differing meshing algorithms.

Absolutely any change made to the model of the electrical machine can be reverted and any part of the electrical machine can be replaced with an equivalent part.

1.7 To Complete The Electrical Machine

Our example slot, figure 1.12 has been divided up into a number of mesh tiles through the addition of several circle and line segments, construction lines as we call them. The intersection between these line segments allows convenient alignment of discrete segments that define the outlines of mesh tiles, placed such that they separately mesh areas corresponding to differing materials.

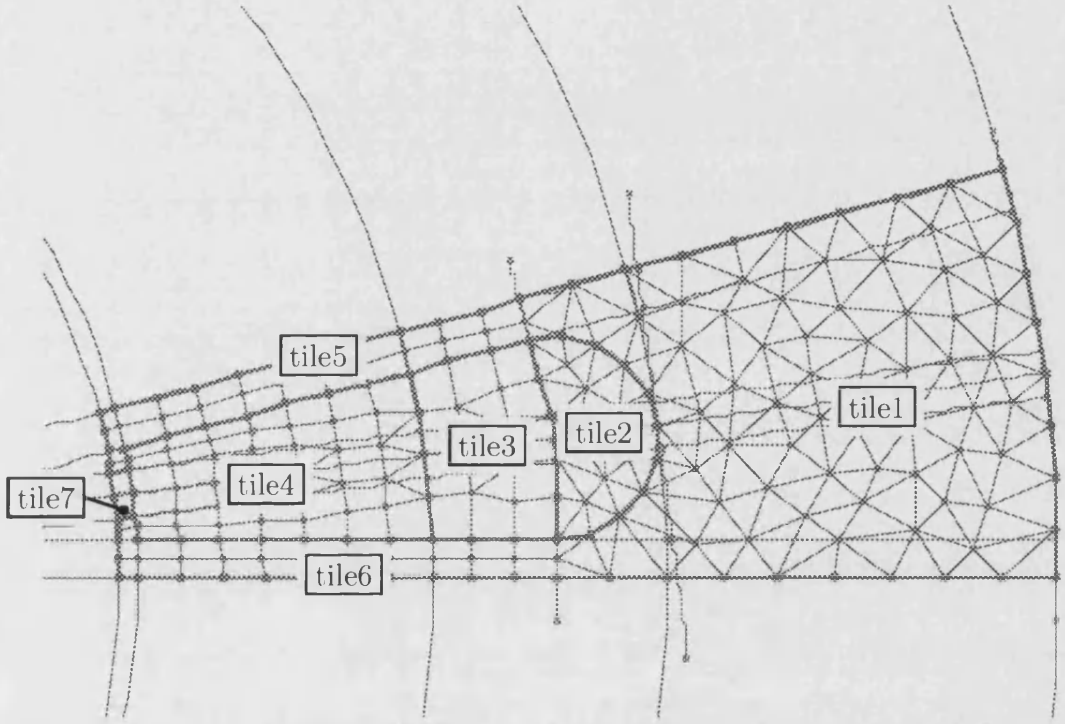


Figure 1.12: Here's One I Meshed Earlier

The grouping of mesh tiles then defines regions, each region taking an index into a table of region properties:

```
AIR = region(2, tile7)
LAMINATIONS = region(4, tile1, tile5, tile6)
TOP_COIL = region(6, tile4)
BOTTOM_COIL = region(7, tile2, tile3)
```

Whilst we have divided our area into more mesh tiles than required, so as to achieve greater control over the meshing, it's then possible to combine any number of these tiles into a single region. Figure 1.13 shows we have divided the slot into

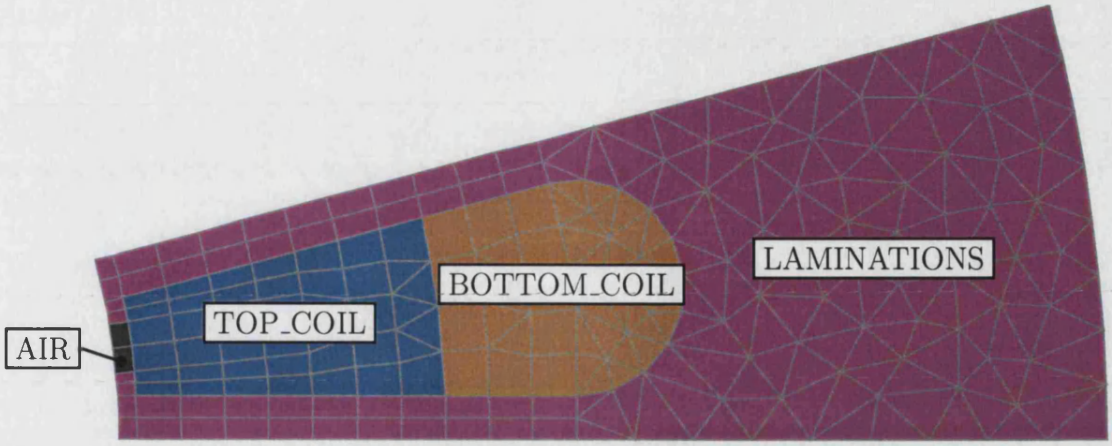


Figure 1.13: Mesh Tiles Now Combined Into Regions

two separate regions of equal areas so we can define two separate coils per slot. To produce the entire stator we need a geometric mapping of elements that will rotate this geometry through *a2* degrees, the angle of one slot, as seen back in figure 1.3:

```
geomap = rotate(ctr, 0, 0, a2)
```

Geometric mappers simply take a coordinate and translate it, the above mapper allows rotation about a point, we'll use our origin *ctr*, rotating through the x, y, and z axis respectively. We will also need to map the coil regions to accommodate different coil arrangements in the slots. The simplest method of achieving this is to use the region map editing tool to establish a region mapping object that corresponds to table 1.1 which defines the polarity and phase of each slot winding. This table lists our two stator coil regions along with their indexes into the region property table, these indexes are arranged by *part*, not slot, the reason being that we are not restricting the use of this mapping to rotating machines. The geometric mapper can be swapped for one with a linear translation so we can then apply our design technique to linear machines. We determine what constitutes a part with the final stage, the component.

Region	Part							
	0	1	2	3	4	5	6	7
TOP_COIL	RPOS (6)	YPOS (7)	BPOS (8)	RNEG (9)	YNEG (10)	BNEG (11)	RPOS (6)	YPOS (7)
BOTTOM_COIL	YPOS (7)	BPOS (8)	RNEG (9)	YNEG (10)	BNEG (11)	RPOS (6)	YPOS (7)	BPOS (8)

Region	Part							
	8	9	10	11	12	13	14	15
TOP_COIL	BPOS (8)	RNEG (9)	YNEG (10)	BNEG (11)	RPOS (6)	YPOS (7)	BPOS (8)	RNEG (9)
BOTTOM_COIL	RNEG (9)	YNEG (10)	BNEG (11)	RPOS (6)	YPOS (7)	BPOS (8)	RNEG (9)	YNEG (10)

Region	Part							
	16	17	18	19	20	21	22	23
TOP_COIL	YNEG (10)	BNEG (11)	RPOS (6)	YPOS (7)	BPOS (8)	RNEG (9)	YNEG (10)	BNEG (11)
BOTTOM_COIL	BNEG (11)	RPOS (6)	YPOS (7)	BPOS (8)	RNEG (9)	YNEG (10)	BNEG (11)	RPOS (6)

Table 1.1: Mapping Coil Regions Across Slots

1.8 Components

A component is intended to exploit the symmetry within electrical machines so that only part of the machine need be constructed. In rotating machines a logical part to construct is the slot. If we use a component to combine the four regions of figure 1.13 into a single part, the component can produce the entire stator of our electrical machine. The component needs to be told how to geometrically map these regions to produce a second part, we've already created a rotating mapper for this purpose. If it's told to repeat this process it will take the second part and map this to create a third, and so on as required. The syntax that will create our component is as follows:

```
stator = component(geomap, N, stator_region_map,
                   AIR, TOP_COIL, BOTTOM_COIL, LAMINATIONS)
```

We've provided the component with the geometric mapper, a number of repetitions, a region mapper, and finally the list of regions contained. If the number of repetitions is set to zero the component does no mapping, it contains only the regions given and forms a single part denoted as *part 0*. With a little parameterisation we can play around with the geometry of the machine on a grand scale. The following sets the basic angular geometry according to a single parameter, the number of slots:

```
slots = 24
a2 = 360 / slots
repetitions = 360 / a2 - 1
stator = component(geomap, repetitions, stator_region_map,
                   AIR, TOP_COIL, BOTTOM_COIL, LAMINATIONS)
```

Figure 1.14 identifies a few of the mapped parts and their regions from the final mesh for correlation against table 1.1.

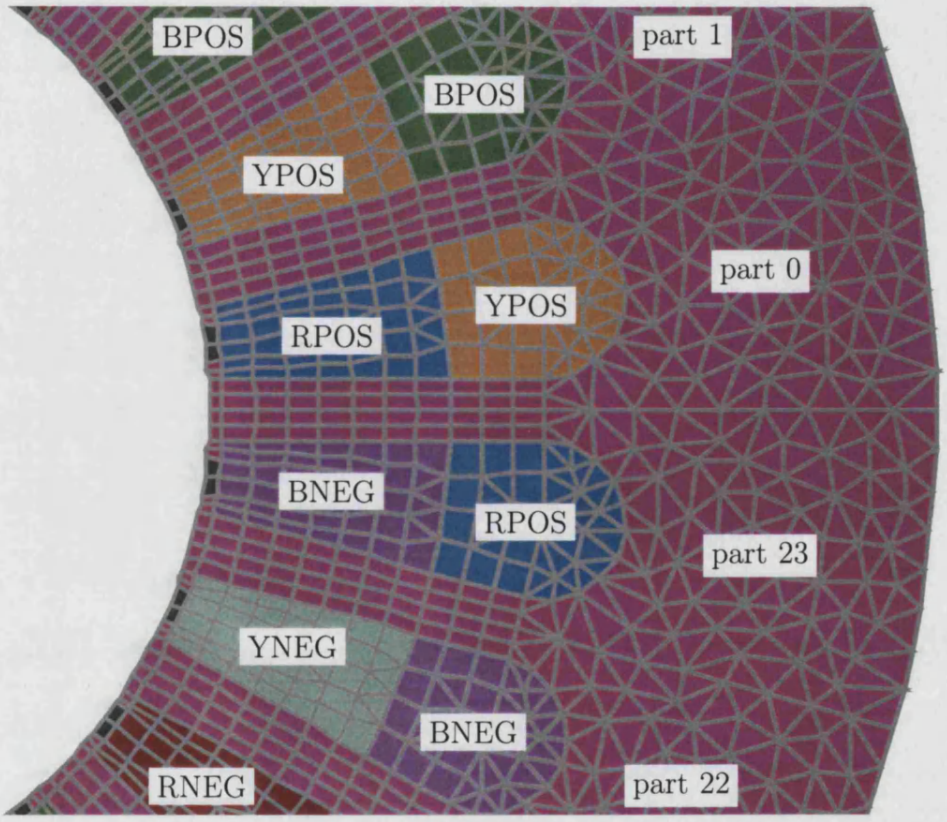


Figure 1.14: Component Mapped Parts and Regions

1.9 Libraries Of Components

However complex the geometry of a rotating machine's part becomes, all parts are still fundamentally defined by a centre point, two diameters, and two angles, *ctr*, *d1*, *d2*, *a1*, and *a2* respectively, so they're interchangeable. Parts can therefore be exported as library components and so a new machine can be constructed simply by importing the relevant parts. The import process recognises parts of a rotating machine and prompts for the number of slots, internal diameter, and external diameter. The import can then setup the basic parameterisation for the machine as we've seen before:

```
slots = <set from input>
a2 = 360 / slots
geometric_map = rotate(ctr, 0, 0, a2)
repetitions = 360 / a2 - 1
```

The program then takes you through the region mapping, defining the region properties at the same time. Once complete the geometric mapping and region mapping exists to construct a revised component for the new electrical machine.

1.10 Benefits Of An Object Oriented Approach To Electrical Machine Design

- We have broken the electrical machine down into separate objects, its building blocks, such as segments and mesh tiles. Within these families of objects everything is interchangeable so we can refine part of the mesh through the interchange of mesh tiles and we can experiment with different coil winding arrangements through different region mappings. Each change can be as simple as the substitution of one region mapper for another but the time saved, from having to redefine the identity of tens of regions, can be considerable.

- Object oriented programming has for example allowed us to easily provide a family of segments that are completely interchangeable. Thus no single

line, arc, or circle segment knows about any other type of segment. An additional advantage of this transparency is in the ability to easily expand the program. If no segment specifically knows about any other segment then we can add to the family of segments without fear of breaking those that already exist or the program. The author has on occasion found the need for a different definition of arc or line segment in order to model a previously unencountered geometry in the most convenient form. The addition of a new arc segment simply needs the computational routine to calculate its geometric values, three points that define its position. One only has to look at a CAD package[5] to see the numerous ways of drawing the same graphics primitive, the building blocks of the machine can be supplemented at a later day with ease.

- All the building blocks describing the electrical machine are stored in a dependency tree framework. A general tool swaps objects in and out of the dependency tree without requiring any specific knowledge of the objects being handled. In object oriented terms this is achieved by having all building blocks of the machine inherit a dependency tree object. The opposite of this is also true, no single building block of the electrical machine needs any specific knowledge of how to be added, replaced, or removed from the dependency tree. This provides us with a generic undo and redo mechanism. Any change made to the electrical machine is reversible, the changes can also be rewound backwards and forwards. Any building block additions cannot break this mechanism and undo information no longer needs to be programmed into every building block[12].

- A dependency tree clearly defines the relationship between objects. Constraints work down the tree and cannot work upwards, when changes are made to the parameterisation the values concerned filter changes down the tree and there's no need to solve simultaneous equations[13] that balance changes between parameters in order to calculate new geometry. This not only simplifies the design but allows changes to be made very quickly. The time taken is linear, the time taken for the change to filter down from the top of the tree to the bottom.

- Through having a descriptive language for representing the electrical machine, rather than just a list of nodes and elements, we are less dependent on graphical tools that can manipulate our design. In making changes to a machine the author has found that a textual search and replace on the

machine's description has been as powerful asset, or compensated for the lack of, more sophisticated graphical tools in the demonstration program. Using this language a complete mesh refinement can be achieved by a single statement altering the parameterisation; we can create optimisation loops which iterate through different parameter values in order to find the one yielding the best result.

- The parameterisation this scheme offers allows the reuse of designs by initially defining parts of the electrical machine, such as the slot, in simple geometric terms. When commencing the design of an electrical machine these parameters allow past designs to be imported and slotted into place in the new machine. In this thesis we will demonstrate this re usability as the stator of a high speed motor, with permanent magnet rotor, is reused to form an induction motor.

1.11 The Structure of This Report

Chapter 2: Modelling An Electrical Machine introduces the individual components we have identified to construct the finite element mesh of an electrical machine.

Chapter 3: Communication Between Objects details the dependency object, used as a basis for the components describing an electrical machine, which gives the ability to refine our finite element mesh through the parameterisation of its defining parts.

Chapter 4: Building Objects, Pre-processing takes us from the input definitions of the machine to prototypes, the mechanism that allows us to keep adding electrical machine building blocks without having to modify any of those already existing.

Chapter 5: Building details a construction mechanism that is independent of the types of object being constructed, that allows us to replace objects within our design with different objects i order to redesign the machine.

Chapter 6: Build Post-processing shows how we can apply specific post-processing to the design of an electrical machine, allowing us to tailor the mechanism to a particular application.

Chapter 7: Building Blocks of an Electrical Machine gives greater detailed descriptions of the geometric objects we have used to describe the electrical machine, resulting in the mesh used for finite element analysis.

Chapter 8: Conclusions are drawn from the work.

Chapter 9: Future Work a brief list of topics further developing this work.

Chapter 2

Modelling an Electrical Machine

The object oriented approach to electrical machine design identifies the separate objects, or building blocks, that comprise an electrical machine and provides a flexible framework in which these building blocks can be put together. This framework has many advantages for the designer in terms of the ease in which a model can be constructed and altered, along with the time saved through this design approach which encourages the reuse of designs.

With these building blocks it should, the author hopes, be possible to 2-dimensionally model and provide the mesh representation for any electrical machine, ready for solution using finite element analysis.

In this chapter we identify the building blocks of an electrical as we work through the construction of a real world example of a high speed motor.

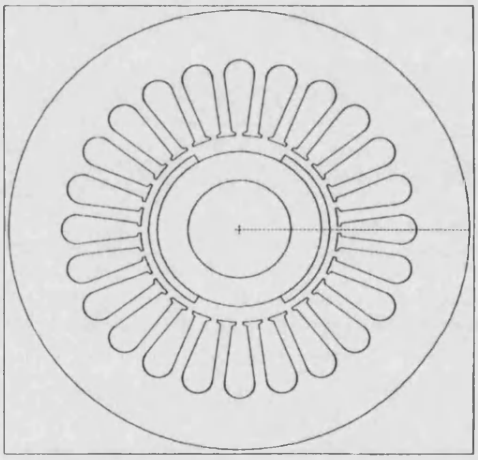
Figure 2.1 shows the specification sheet of a high speed motor with a 3-phase wound stator and permanent magnet rotor. This chapter focuses on modelling the stator of this machine in a way that allows the design to be reused at some point in the future.

1- Dimensions

Lamination Outside OD = 135 mm
Lamination Inside Diameter = 57.5 mm
Stack Length = 80 mm
Tooth width = 3.5 mm
Slot Depth (From stator bore to slot bottom) = 23.25 mm
Tooth tip thickness = 0.75 mm
Slot openning = 2.0 mm
Number of slots = 24
Magnet thickness = 2.5 mm
Magnet angle-expansion = 120 deg
Rotor diam at outer magnet surface =52.85mm
Clearance gap between outer magnet surface and inner stator surface =2.323mm

2 - Materials

Stator core made of SiFe with a thickness of 0.2 mm
Magnet material Br = 1.05 T; Hc = 79 kA/m; Ur = 1.05



3- Control Data

Supply voltage at motor terminal = 560 V DC
Current Limit set to 170 A
Motor speed = 80000 rpm
Control is set to be Sinusoidal control

4- Winding Data

Number of turns = 1 turns
Number of strands in hand for one conductor = 40
Slot fill on bare diameter = 0.4
Wire diameter = 0.79 mm
4s/p/p with turn span of 9 slots
Winding factor = 0.886
Phase resistance = 4.9 E-03 Ohm
Phase Self Inductance = 0.130 mH
Mutual Inductance between phases = 0.046 mH

Figure 2.1: Data For The High Speed Motor

2.1 Basic Parameterisation That Allows Reusable Machine Parts

Whilst the designer needn't restrict themselves to any specific parameterisation in order to describe an electrical machine, the following parameters have been chosen to describe part of a rotating machine so that parts from different machines become interchangeable through this standardisation. The standard part for a rotating machine is the slot, which figure 2.5 shows can be described by a centre point, *ctr*, internal diameter, *d1*, external diameter, *d2*, an angle, *a1*, that describes the angular starting position of the slot, and angle, *a2*, that describes the angular extension of the slot. The centre and starting angle of the slot are unlikely to need adjustment, and thus arguably any parameterisation, but parameterisation of these values is straightforward and offers further potential flexibility.

2.1.1 Values, The First Building Block And The Parameterisation Of The Electrical Machine

Our electrical machine building blocks fall into families, figure 2.2, one such family are the values which extend from simple floating point numbers through to the dot product of two vectors. Each value is interchangeable with another value so the angular value *a2* could be fixed:

```
a2 = 15
```

or it could be the result of an arithmetic expression:

```
a2 = 360 / slots
```

Ultimately even the simplest of other building block families, such as points, will depend on values to fix their coordinates for instance, so values always set the parameterisation of a machine.

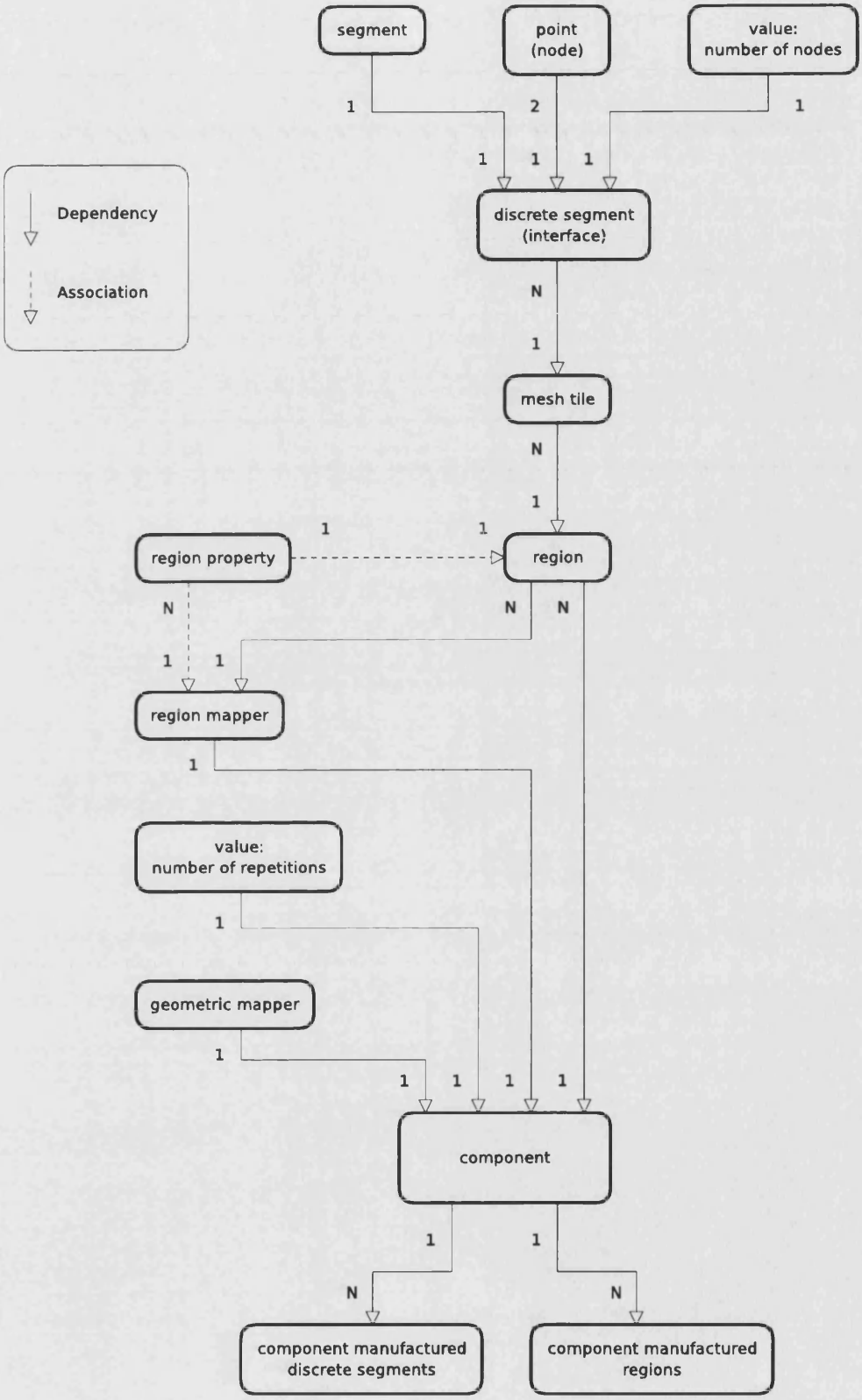


Figure 2.2: Relationship Between Different Families Of Object

2.1.2 Assigning Values

Data entry is of the form:

```
<variable name> = <some value or other building block>(arguments)
```

This data is interpreted by a pre-processor, chapter 4, which understands many short cuts that allow the likes of:

```
x = real(0)
y = real(0)
ctr = vvpoint(x, y)
```

to actually be entered as:

```
ctr = vvpoint(0, 0)
```

Here two substitutions are being performed by the pre-processor. It's first recognising that two assignments are real numbers and expand to the *real* type of value:

```
ctr = vvpoint(real(0), real(0))
```

It subsequently recognises the nesting of assignments, separating them out and manufacturing names so the above becomes:

```
_vx_ctr = 0
_vy_ctr = 0
ctr = vvpoint(_vx_ctr, _vy_ctr)
```

Each value and point consists of a separate object residing within a dependency tree. One exception to this is the *infix* value which allows flexible input of mathematical expressions such as:

```
a2 = 360 / 24
```

The pre-processor recognises this as:

```
a2 = infix(360 / 24)
```

and the infix prototype [1], which is responsible for building the infix value, will construct several values to satisfy this expression. The dependency tree of figure 2.3 shows the final representation.

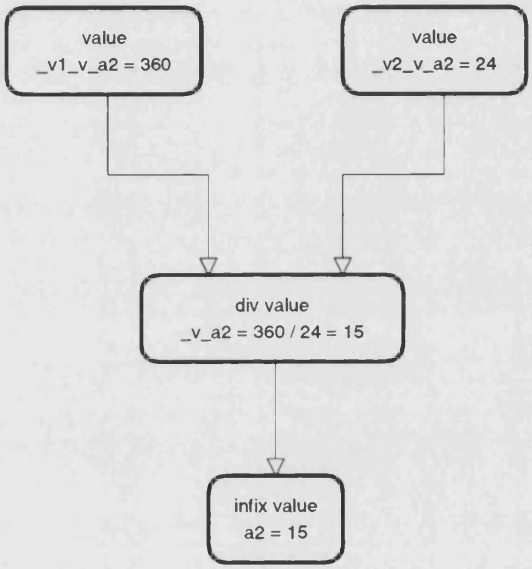


Figure 2.3: Corresponding Dependency Tree For The Infix Value of *a2 = infix(360 / 24)*

2.2 On With The Machine

We lay down the basic parameterisation as follows:

```
d1 = 57.5
```

[1]Each building block of the machine has a prototype, chapter 5, whose responsibility is to ensure the correct arguments are supplied for the object to be constructed. It allows unlimited additions to the program with each addition adding a building block and its prototype. To know if something can be built you look for a like named prototype, being separate entities there's no possibility of an addition breaking the existing program.

```
d2 = 135
a1 = 0
a2 = 360 / 24
```

We've touched on values and points, we now use members of the segment family to draw line, arc, or circle segments which define the outline of our machine. We first draw two circles, both centred at *ctr* with one at the inner diameter of the machine and the other at the outer diameter:

```
r1 = d1 / 2
r2 = d2 / 2

cc1 = pvcircle(ctr, r1)
cc2 = pvcircle(ctr, r2)
```

We then draw two lines that originate from the centre point, *ctr*, one at an angle of *a1* degrees and the other at *a1 + a2* degrees. We ensure these lines are long enough to intersect both circles:

```
cl1 = pvvvline(ctr, _vs_cl1, r2, a1)
cl2 = pvvvline(ctr, _vs_cl2, r2, a1 + a2)
```

The *pvvvline* segment is perfect for this purpose, figure 2.4, it's easily defined by the parameterisation we've started with.

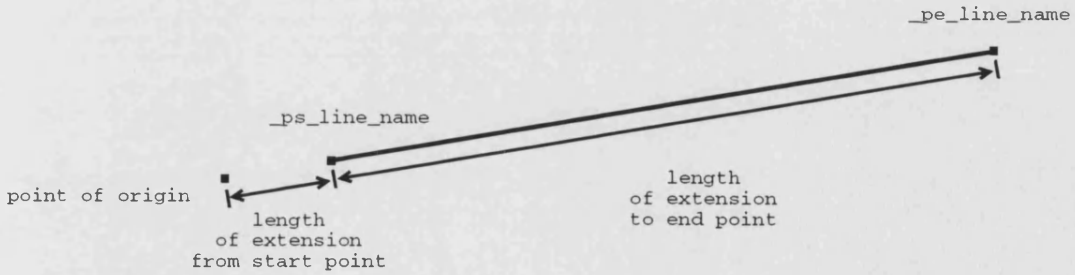


Figure 2.4: Line Segment Used For pvvlines *cl1* And *cl2*, Formed From A Point Of Origin, Start And End Extension, Plus Angle Of Trajectory

The post-processor, chapter 6, discovers where segments intersect, it creates intersection points whose position depends upon the segments and the point they

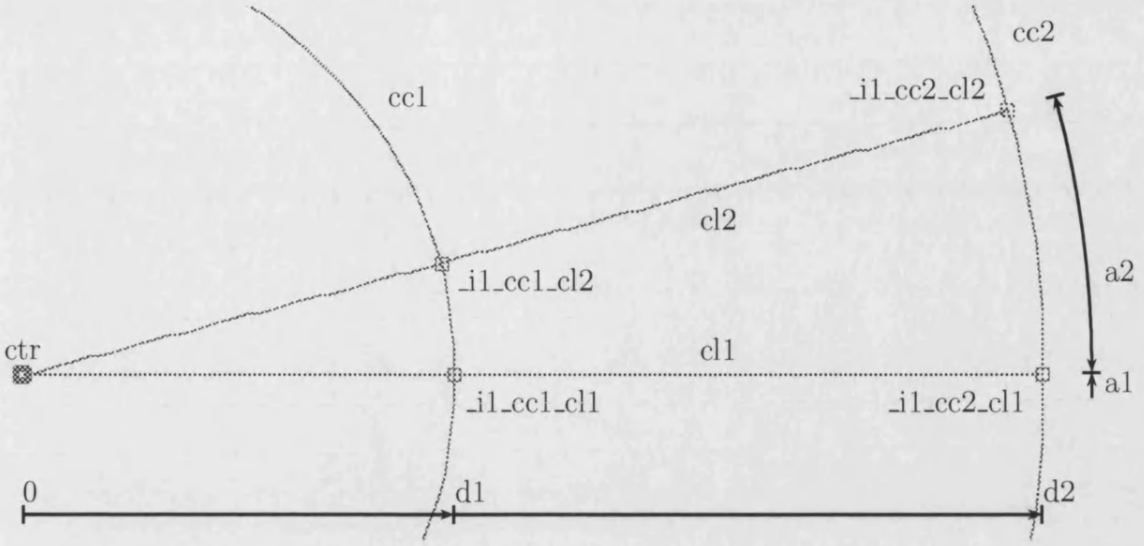


Figure 2.5: Laying Down The Fundamental Parameterisation Of A Stator Slot

intersect. If we change our parameterisation so that these segments move position, intersection points will move accordingly. We've already created segments that depend on the centre point, any line segments that depend on the intersection points will accordingly move as parameterisation changes filter down the dependency tree.

The outline of the slot has now been formed, figure 2.5, the parameterisation allows this rotating part to be slotted into place in the stator of any rotating machine.

Our parameterisation doesn't end here, however any further parameterisation will not affect this part's ability to be imported into other designs. Additional parameterisation will be part specific, serving only to increase the flexibility of a particular design.

2.2.1 Parameterisation Of Slot Teeth

Parameterisation will now be introduced allowing variation in the width of the slot teeth.

First we define some additional parameters, taken from the specification of the high speed motor:

```
slot_depth = 23.25
slot_opening = 2
tooth_tip_thickness = 0.75
tooth_width = 3.5
```

We then place another circle, *cc0*, inside the circle *cc1* whose radius is set to *r1*.

```
cc0 = pvcircle(ctr, r1 * 3/4)
```

The intersection of this circle and the construction lines *cl1* and *cl2* will be used to anchor several construction lines that won't directly form the opening of the slots. Keeping these segments away from the workspace surrounding the slot opening will prevent this area from appearing cluttered and will facilitate later construction.

Line segments can now be placed where the construction lines *cl1* and *cl2* intersect this new circle and the larger circle, *cc2*.

```
cl001 = spvvvline(cc0, _i1_cc0_cl1, tooth_width/2, tooth_width/2, 0)
cl002 = spvvvline(cc0, _i1_cc0_cl2, tooth_width/2, tooth_width/2, 0)
cl201 = spvvvline(cc2, _i1_cc2_cl1, tooth_width/2, tooth_width/2, 0)
cl202 = spvvvline(cc2, _i1_cc2_cl2, tooth_width/2, tooth_width/2, 0)
```

We use a line segment called the *spvvvline* [2] for these anchors, illustrated in figure 2.6. This line segment produces a line parallel to a straight line, or the tangent of an arc or circle, at a given point. The new line extends out from either side of the reference point, like a seesaw, the extension either side being adjusted independently. The line can then be rotated around this point so a value of 90 degrees creates a normal to the original segment. Figure 2.7 shows the four line segments, *cl001*, *cl002*, *cl201* and *cl202*.

[2]Names used for different building block objects derive from the arguments they are supplied with. The *spvvvline* requires a segment, point, and three values respectively, to draw the line. This naming convention has so far resulted in unique names and alleviated the author from the need to invent memorable and succinct names. Please note that these names can be given any number of aliases which may better describe their function.

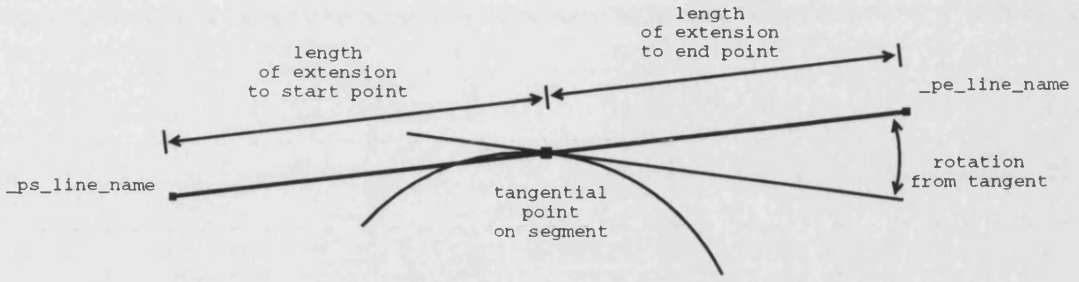


Figure 2.6: Line Segments Used For Forming The Parallel Lines Of The Slot Tooth

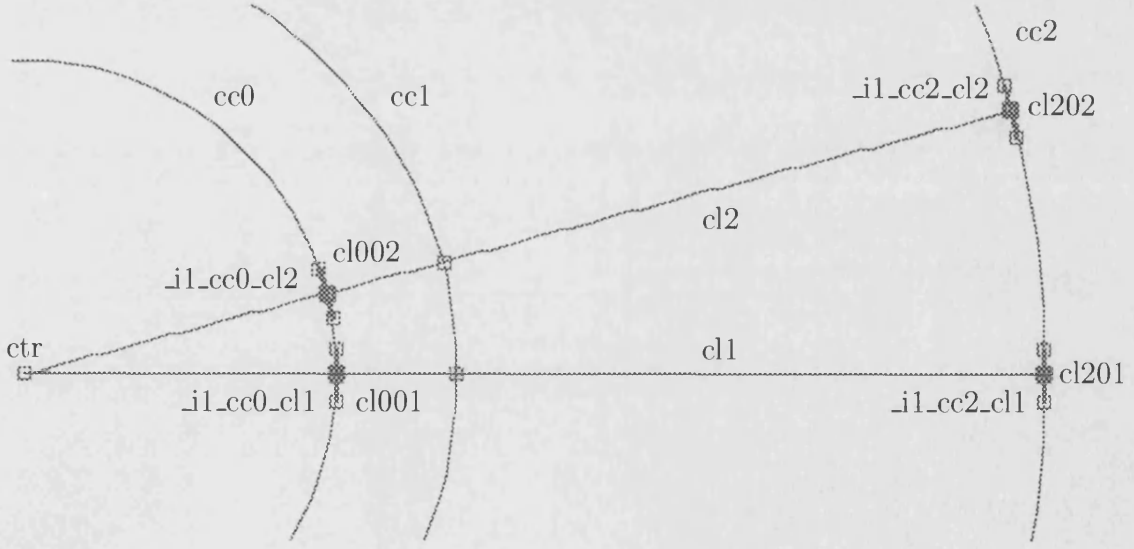


Figure 2.7: Forming Anchors For The Parallel Lines Distanced Of The Slot Teeth

In comparison to the line segment *ppline*, which connects between two points and which we'll next use, the *spvvvline* has no points associated with the ends of its line. Such points are useful as they provide a reference to the end of the line, one which we can always attach another line to for instance. To compensate for this the *spvvvline*, knowing that no such points exist, manufactures points for just this purpose. The points *_ps_cl001* and *_pe_cl001* will be found at the start and end of line segment *cl001* respectively. We use these manufactured points to connect the point to point lines *cl4* and *cl5* which are illustrated in figure 2.8 and defined as follows:

```
cl4 = ppline(_ps_cl001, _ps_cl201)
cl5 = ppline(_pe_cl002, _pe_cl202)
```

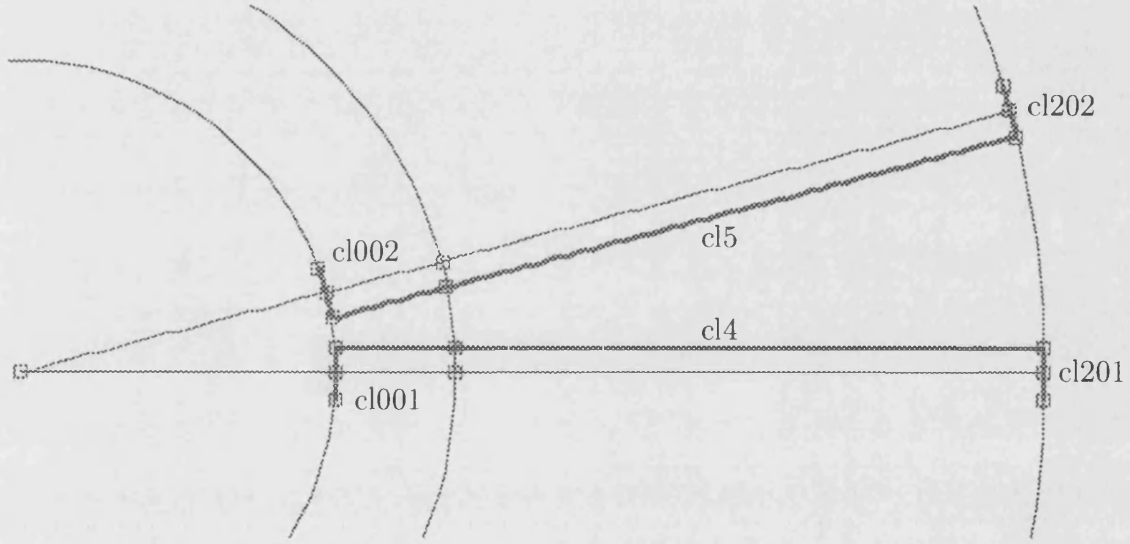


Figure 2.8: Placement Of The Parallel Lines Forming The Length Of The Slot Teeth

2.2.2 Parameterisation Of Slot Depth

Previously we parameterised the depth of the slot:

```
slot_depth = 23.25
```

we now use this value to construct a circle *cc3* which will later seat the base of the slot. The arrangement of figure 2.9 shows the position of this circle and the

construction line *cl3* which divides the slot into two equal halves. Where these two segments intersect we form another *spvvvline* which is parallel to the tangent of circle at this point:

```
cl3 = pvvvline(ctr, 0, ,r2, a1 + a2/2)
cc3 = pvcircle(ctr, r1 + slot_depth)
cl303 = spvvvline(cc3, _i1_cc3_cl3, 10, 10, 0)
```

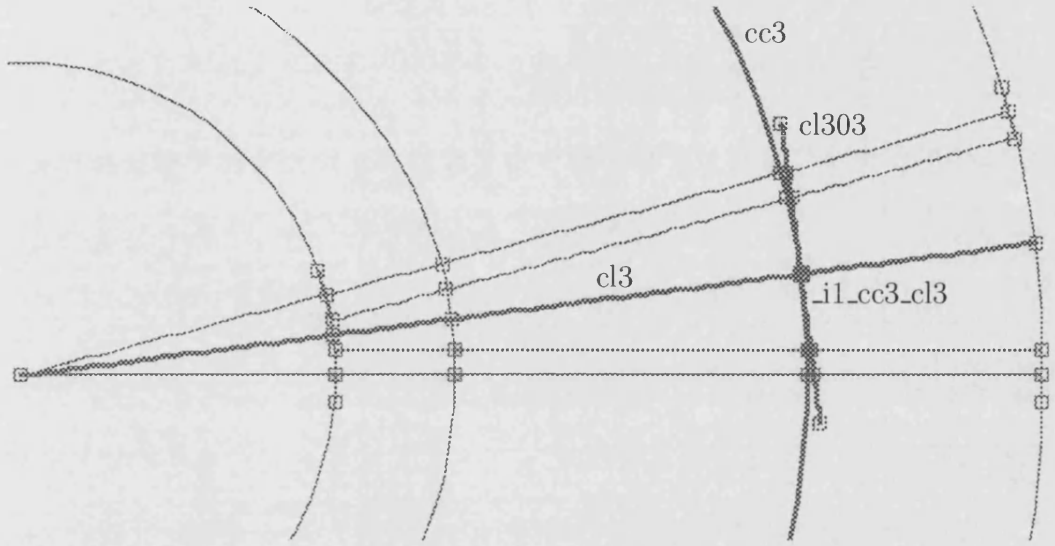


Figure 2.9: Slot Depth Parameterisation Is Used To Set The Radius Of Construction Circle *cc3* Which Will Eventually Anchor The Base Of The Slot

Construction line *cl303* forms the final segment needed to complete the base of the slot. In figure 2.10 four intersection points can be seen, the last two of which were created by the intersection of *cl303* and construction lines *cl4* and *cl5*. Joining these points together we create the three lines illustrated in bold, the line between the first and last points of the argument list is always ignored:

```
ca1 = pppparc(_i1_cc1_cl5, _i1_cl303_cl5, _i1_cl303_cl4, _i1_cc1_cl4)
```

The arc is formed from the largest circle that can be enclosed by this area whilst still touching the three specified sides. The lines will be tangential to the arc at each point of contact. The contact points between the arc and two side lines set the start and end of the arc.

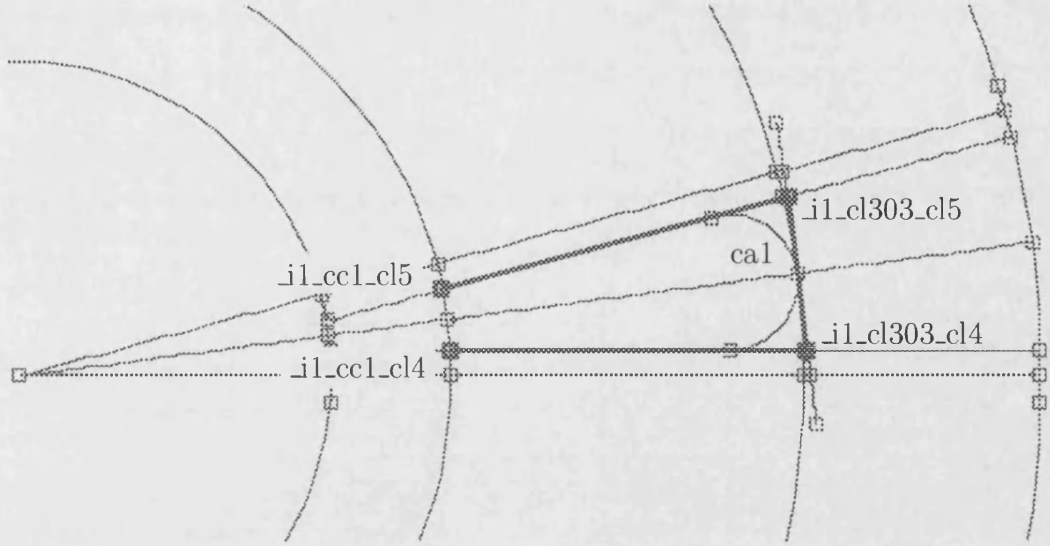


Figure 2.10: Base Of Slot Completed By An Arc That Fits Itself Within Three Vectors Formed By Our Construction Segments

2.2.3 Parameterisation Of Slot Opening

The outline of the slot is nearly complete, we have yet to form its opening or set the thickness of the slot tooth tip. These are formed with a similar arrangement to that used for the parallel lines of the slot walls.

The *spvvvline* segment that provides the seesaw like line arrangement is used to extend a line outwards from an intersection point, figure 2.11. The length of the line each side of the centre point is half the distance of the slot opening, total length of the line is thus equal to the size of the slot_opening. Lines *cl003* and *cl203* are both done in this manner and lines *cl6* and *cl7* interconnect their end points:

```
cl003 = spvvvline(cc0, _i1_cc0_cl3, slot_opening/2, slot_opening/2, 0)
cl203 = spvvvline(cc2, _i1_cc2_cl3, slot_opening/2, slot_opening/2, 0)
cl6 = ppline(_pe_cl003, _pe_cl203)
cl7 = ppline(_ps_cl003, _ps_cl203)
```

This arrangement has created the slot opening. As the *slot_opening* parameter is varied the lengths of *cl003* and *cl203* will correspondingly change, the line always remaining centred around its anchoring intersection point. Lines *cl6* and *cl7* will

also follow position and their intersection with circle *cc1* forms the slot opening. If we add another circle that is distanced from *cc1* by the *tooth_tip_thickness* then our slot outline is complete, *cc11* also detailed in figure 2.11. The line between the intersection of *cl6* and *cc1* and the intersection of *cl6* and *cc11* forms the tooth tip thickness, mirrored by the respective intersections of *cl7* and these two circles.

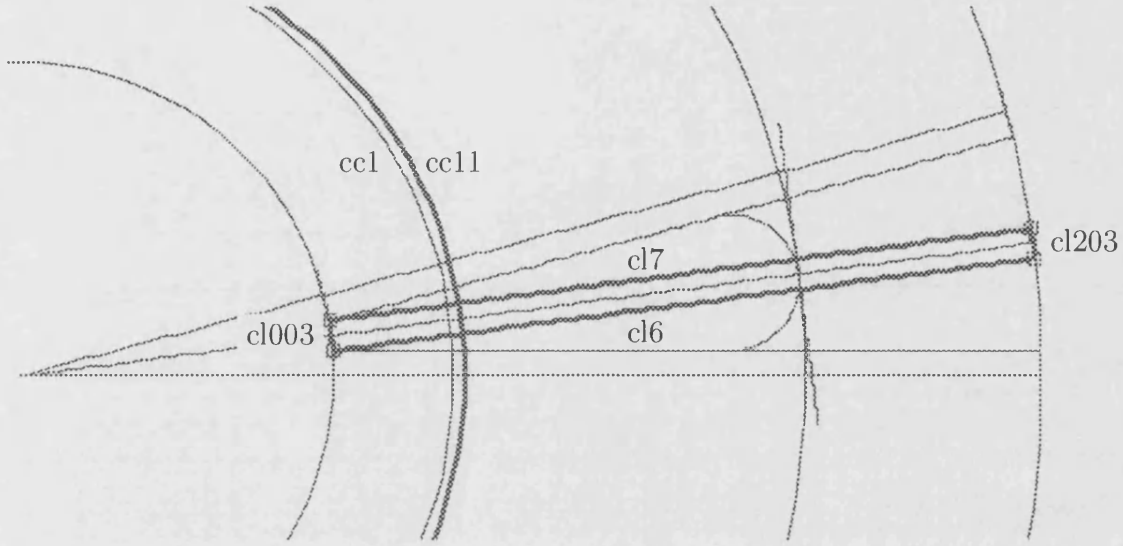


Figure 2.11: Slot Opening, The Radial Difference Between *cc1* and *cc11* Equals The Value Of *tooth_tip_thickness* And The Parallel Lines *cl6* And *cl7* Are Spaced Apart By The Value Of *slot_opening*

2.2.4 Division Of The Slot In Preparation For Coil Regions

Each slot is shared by two coils but the slot area isn't equally divided between these two coils, instead one coil occupies three fifths of the area whilst the other occupies two fifths. In addition to this the division alternates from one slot to the next, first the upper layer of windings occupies three fifths of the area and then the lower layer occupies three fifths of the area in the next slot. Figure 2.29 aids in visualising this arrangement.

For now, all we need do is divide the area of slot from top to bottom into areas of two fifths, one fifth, and two fifths again. The middle fifth of the area will be grouped either with the top two fifths or the bottom two fifths depending on the arrangement of coil windings. Here we cheat, the author has calculated the

necessary divisions outside of the program thus compensating for the lack of a segment type that does this and the time needed to create one. The first construction circle is placed at around 48% of the length of the slot depth, measured from the slot opening, the second division at about 66%:

```
ccr1 = pvcircle(ctr, r1+tooth_tip_thickness+slot_depth*(48.18/100))
ccr2 = pvcircle(ctr, r1+tooth_tip_thickness+slot_depth*(65.67/100))
```

Figure 2.12 shows these construction lines in place.

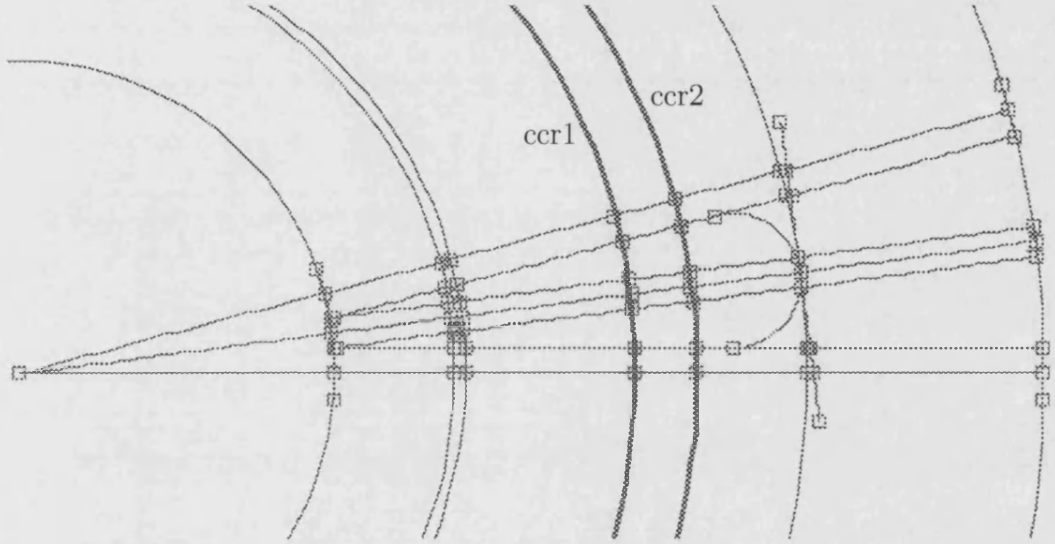


Figure 2.12: Division Of Slot Into Areas Of Two Fifths, One Fifth, And Two Fifths

2.3 Discrete Segments

We now have all the necessary segments in place to be able to outline our slot, and any regions within, by moving along its perimeter from a point to a segment and back to a point again, marking each of these sections as a line of nodes. At this stage we already have some nodes in our model, every point is actually a node but we just haven't designated which ones we are yet to use. Taking the sections along segments, spanned by two points or nodes, we mark these spans as a line of nodes by using the *discrete segment*. Areas contained on all sides by these discrete segments are like tiles that fit together to construct the mesh

of the slot. Each one of these tiles will be meshed individually, the density of the mesh being controlled by the number of nodes along these discrete segments. By splitting the whole of the slot mesh into smaller mesh tiles we have greater control over the mesh density in different areas.

Figure 2.13 shows a discrete segment spanning a construction line segment between two intersection points. The discrete segment is created using the following syntax:

```
n = 1.0

ds1 = dsegment(cl4, _i1_cc11_cl4, _i1_ccr1_cl4, 8n)
```

The first argument supplied is the segment's name, the two subsequent arguments are points which must reside on the segment. The final argument specifies the number of nodes to create along the discrete segment, because the discrete segment connects between two existing points, which are also nodes, the discrete segment only creates nodes within the length spanned by these terminating nodes. This use of existing nodes helps prevent duplicate nodes, another discrete segment attached to *_i1_cc11_cl4* or *_i1_ccr1_cl4* will also reuse their node rather than create a duplicate at the same position. In the above definition of discrete segment *ds1* we have also chosen to parameterise the number of nodes, the value of n could be increased to 1.5 in order to increase the number of nodes to 12. Globally we shall use this parameter with all discrete segments so that the overall mesh density can be controlled.

Creating discrete segments in the above manner can be tedious, identifying segments and points takes the majority of the time. A point and click interface improves this situation by allowing the segment and points to be identified through the graphical interface, however we can simplify the process further by utilising the underlying dependency tree in which or model resides. We find that some segments, such as the *ppline*, have their position defined by two points, other points like intersection points are then defined by two segments; whichever way around the relationship, a connected point and segment will always have a parent child relationship in the dependency tree. The *dsegment* tool utilises this, using the graphical interface points are first selected along the path that we wish to transform into discrete segments. The *dsegment* tool is then invoked and it walks

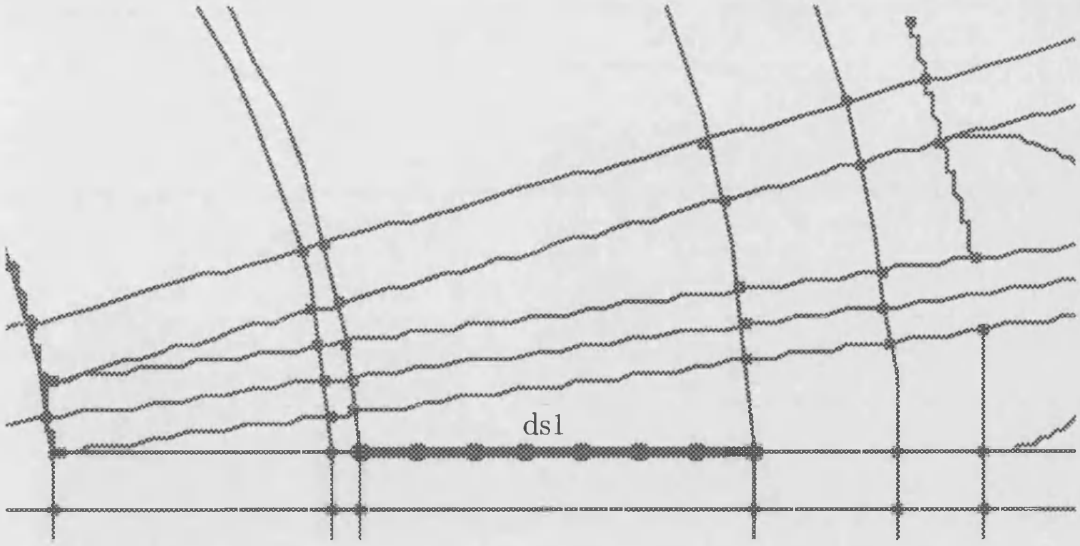


Figure 2.13: First Discrete Segment Begins To Define Node Boundary Of Slot

along a path defined by these points and the segments that interconnect them, at each step the potential discrete segment is highlighted, figure 2.14, and if the designer wishes it created they need only enter an expression for the number of nodes. The entire geometry can be discretised very quickly in this manner.

Figure 2.15 shows all the discrete segments in place and figure 2.16 illustrates them with all construction lines removed and a mesh tile in place. Once all enclosed ares are independently meshed using these tiles we will start to see how they can be combined into regions of the same material. The discrete segments play an even more important role when we start to define regions as they define the interfaces between them, they also define the external interfaces to which we will assign boundary conditions. Just as every point has a node, every discrete segment has an interface. These relationships form the translation from a parameterised geometry into a mesh representation.

2.4 Mesh Tiles

We are now ready to create *mesh tiles* in the area enclosed by discrete segments and have a few meshing algorithms to chose from. The *semesh*, super element mesh, deals very well with simple areas of three or four sides whilst two third part utilities have been interfaced to the program which are capable of more complex

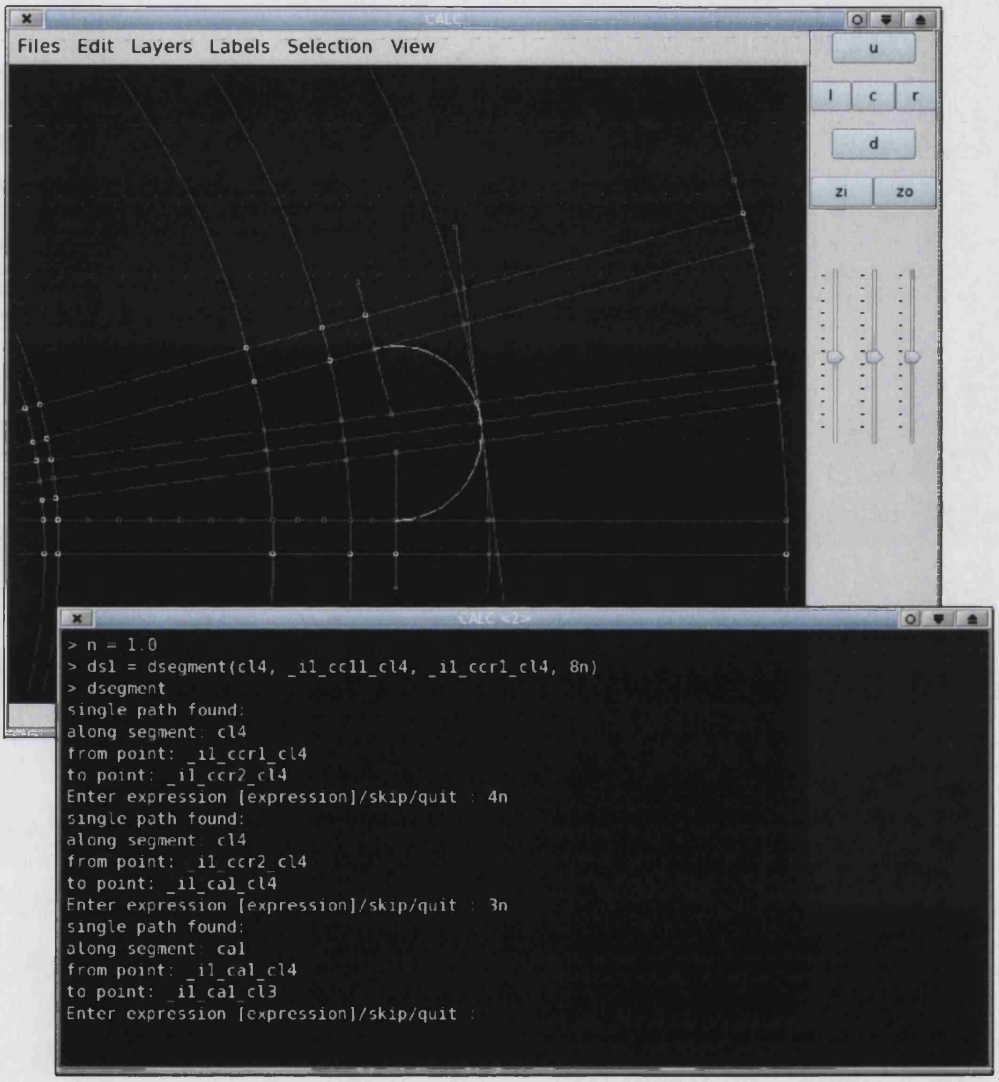


Figure 2.14: Automation Of Discrete Segment Creation, Interactive Path Finder Follows A Trail Of Selected Points Creating Discrete Segments At Each Step

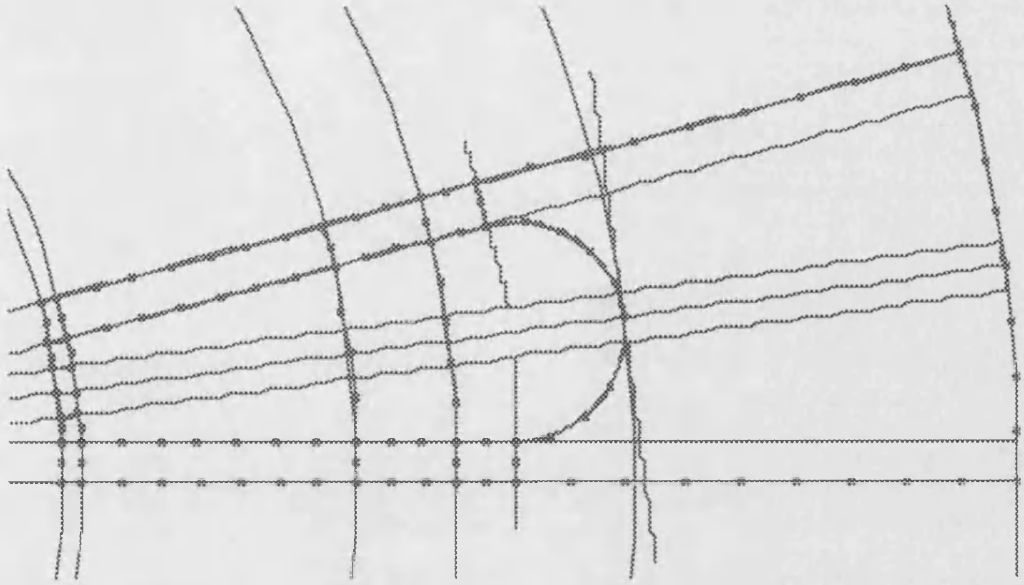


Figure 2.15: All Discrete Segments In Place

geometries. The third party programs, Bojan Niceno's Easymesh[1] and Jonathan Shewchuk's Triangle[2], are separate programs that communicate through files, reading in a list of nodes and writing out a list of nodes and elements. Interfacing to these programs involves writing out the list of nodes, executing the program, then reading in the list of new nodes and elements. This method ensures that should the third party program fail, our program doesn't suffer the same fate; if we don't find valid node and element files on completion then the mesh is put into an invalid state which alerts the designer to a problem.

Using the super element mesh tile we produce the tile *m1* of figure 2.16.

```
m1 = semesh(cds1, ds31, ds72, ds41)
```

In figure 2.17 this tile is then replaced with another tile built using Triangle, reassignment of *m1* to another mesh tile pulls the original from the dependency tree, replacing it with the new tile before storing the original in an undo buffer. At the same time we tile the larger area of our slot using the same meshing algorithm:

```
m2 = triangle(ds20, ds18, ds19, ds16, ds17, ds15)
m1 = triangle(cds1, ds31, ds72, ds41)
```

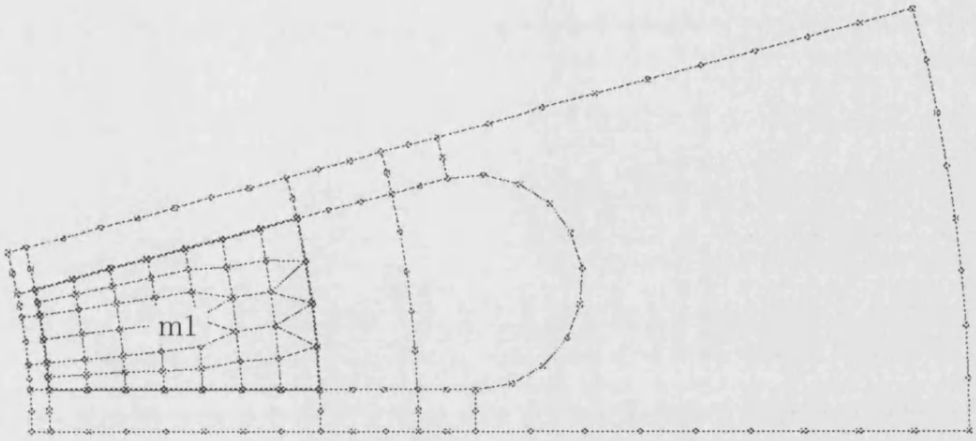


Figure 2.16: Super Element Mesh Tile

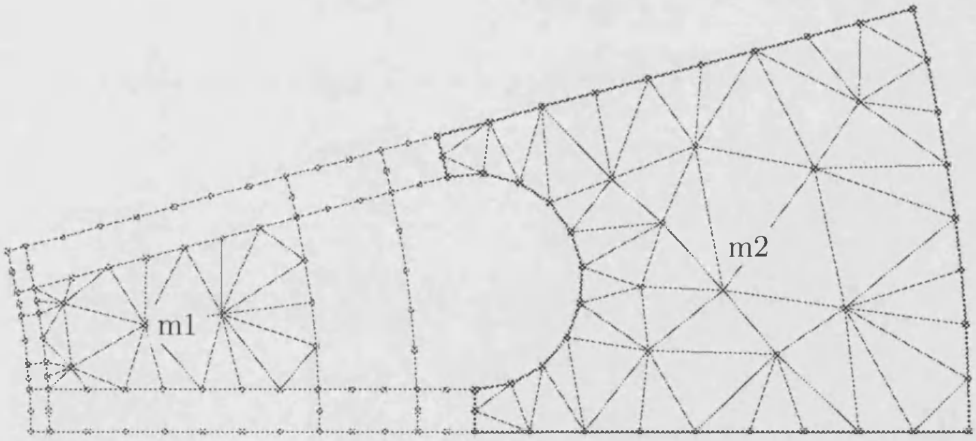


Figure 2.17: Mesh Tiles Built Using An Alternative Meshing Algorithm, Triangle[2]

The larger area would need to be broken down into smaller tiles if we were to use the super element mesh tile, this area we'll leave meshed using Triangle. The smaller area we'll revert to the previous super element mesh tile, the *undo* command will revert the last change to the model and successive undo commands can be used to revert all changes back to the initial state of the model.

Triangle also allows us to restrain the minimum angle and maximum area it uses with elements, we can restrain one and, or, the other and in figure 2.18 the maximum area of elements has been restrained by redefining mesh tile *m2*:

```
m2 = triangle(ds20, ds18, ds19, ds16, ds17, ds15, 0, 5)
```

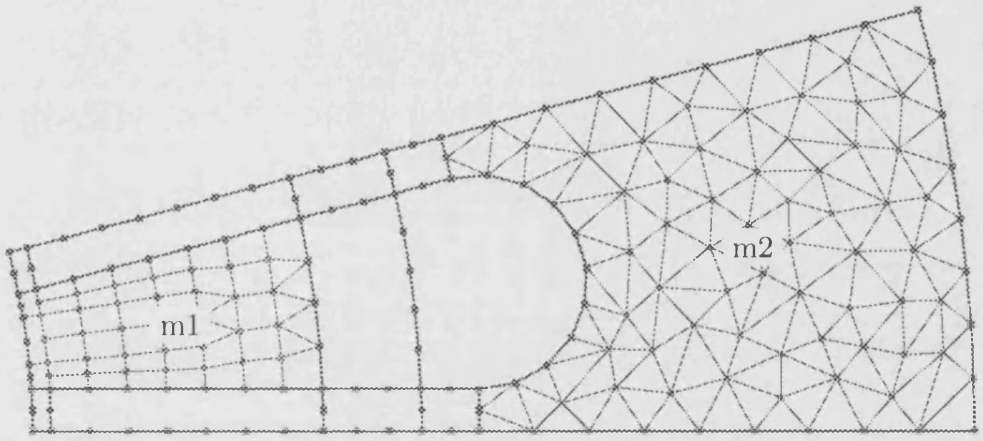


Figure 2.18: A Super Element Mesh Tile and Triangle[2] Mesh Tile With Maximum Area Of Elements Restrained

We complete the slot mesh in figure 2.19, ready for grouping mesh tiles into regions.

2.4.1 Regions

Regions simply group mesh tiles together, they have an index into the region property table so they're also responsible for marking elements with their region identity. Let us define a region, such as the laminations that form the body of our stator slot. This region includes the vast majority of our mesh tiles and we'll numerically identify it with region number five:

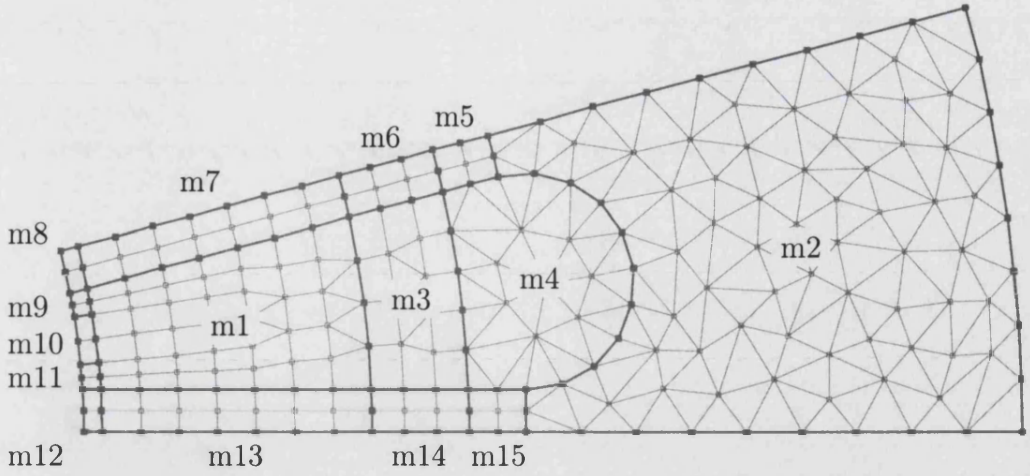


Figure 2.19: All Areas Tiled, Ready For Grouping Into Regions

```
LAMINATIONS = region(5, m2, m5, m6, m7, m8, m9, m11, m12, m13, m14, m15)
```

This adds a little colour to our output of figure 2.20 to aid visualisation. Mesh tiles can be listed in any order in the region's argument list, however the region examines these tiles as it puts them together and it removes any of the discrete segment interfaces found between adjacent tiles. When asked for a list of its interfaces the region now lists only those that are external.

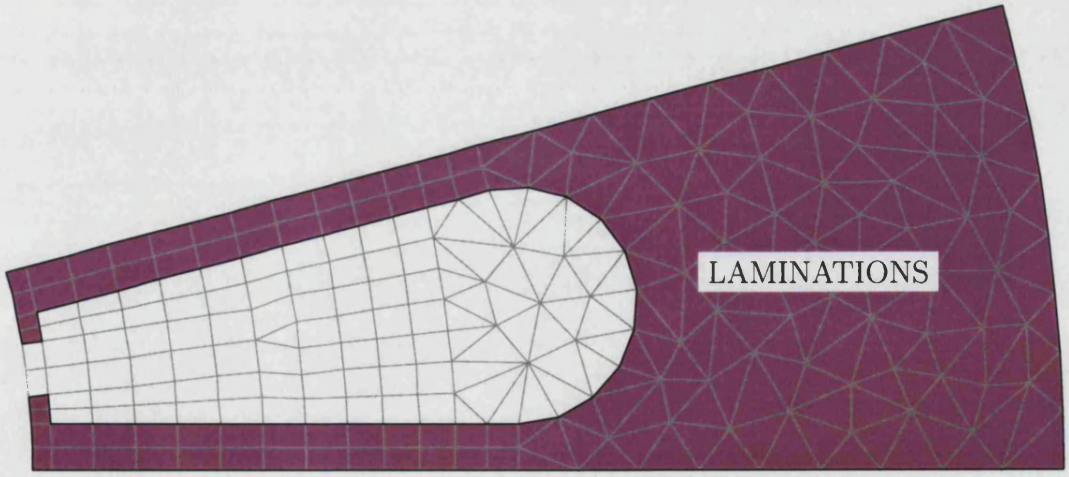


Figure 2.20: Mesh Tiles Grouped By A Region

We can now go into the region properties, figure 2.21, where we'll find an "unset" entry has been automatically created for us. The entry is held in the model as

part of the dependency tree, as is any other building block of the machine. The edit regions tool simply automates the modification of these objects by providing an interactive interface that later creates new region property objects along with the rather long argument list they take. The designer can make unlimited changes in this tool before finally quiting it, these changes are then translated into new region property objects that are stored in the model. Undoing the model at this stage reverts it to the state it was in prior to the running of the region editor.

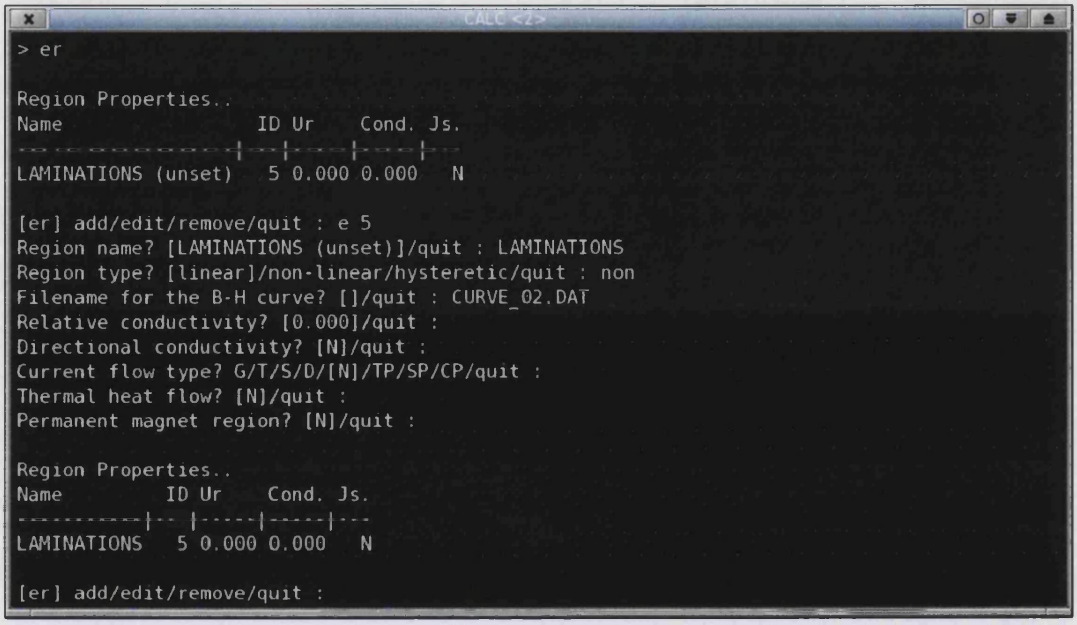


Figure 2.21: Editing The *LAMINATIONS* Region Properties

We can create the small region of air, within the mouth of the slot opening, in exactly the same way, figure 2.22 shows the editing of its region properties.

```
AIR = region(2, m10)
```

2.5 Mapping Regions

Now we come to the more interesting coil regions. If we create the following regions for the top two fifths of the slot area, middle fifth, and bottom two fifths respectively:

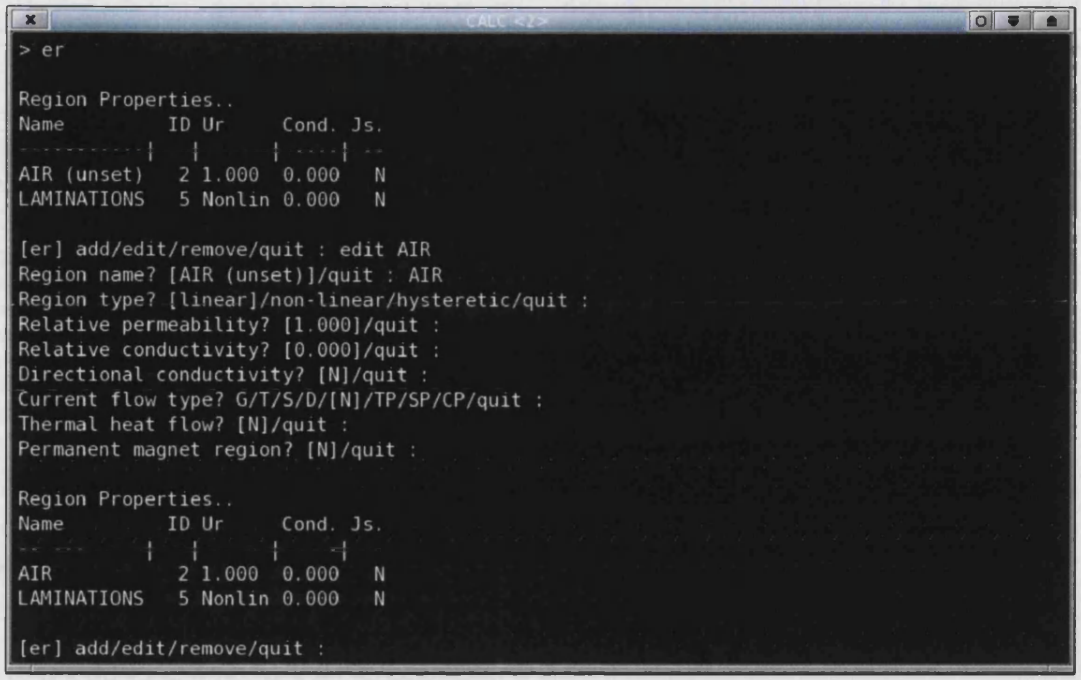


Figure 2.22: Editing The Region Properties Of Region *AIR*

```
TOP_LAYER = region(6, m1)
MIDDLE_LAYER = region(6, m3)
BOTTOM_LAYER = region(6, m4)
```

region number 6 is going to correspond to the positive current of the red phase of our 3-phase winding, the whole of slot 1 contains conductors with this current flow. Slot 2 is split, the top top two fifths contain the positive red phase but the bottom fifth contains the negative blue phase. Table 2.1 shows the current flow of the conductors in each slot and it also shows how we are to assign the above three regions on a per slot basis.

Currently we have only one slot, figure 2.23, which represents slot 1 in table 2.1. Shortly we will create a *component* which constructs the entire stator from the one slot, replicating it twenty three times to produce the twenty four slot machine. However we don't want to assign regions another twenty three times, its tedious and doesn't give us much flexibility if we wish to experiment with different winding configurations.

Region	Slot							
	1	2	3	4	5	6	7	8
TOP_COIL	RPOS (6)	RPOS (6)	RPOS (6)	RPOS (6)	BNEG (11)	BNEG (11)	BNEG (11)	BNEG (11)
MIDDLE_COIL	RPOS (6)	RPOS (6)	BNEG (11)	RPOS (6)	BNEG (11)	BNEG (11)	YPOS (8)	BNEG (11)
BOTTOM_COIL	RPOS (6)	BNEG (11)	BNEG (11)	BNEG (11)	BNEG (11)	YPOS (8)	YPOS (8)	YPOS (8)

Region	Slot							
	9	10	11	12	13	14	15	16
TOP_COIL	YPOS (8)	YPOS (8)	YPOS (8)	YPOS (8)	RNEG (7)	RNEG (7)	RNEG (7)	RNEG (7)
MIDDLE_COIL	YPOS (8)	YPOS (8)	RNEG (7)	YPOS (8)	RNEG (7)	RNEG (7)	BPOS (10)	RNEG (7)
BOTTOM_COIL	YPOS (8)	RNEG (7)	RNEG (7)	RNEG (7)	RNEG (7)	BPOS (10)	BPOS (10)	BPOS (10)

Region	Slot							
	17	18	19	20	21	22	23	24
TOP_COIL	BPOS (10)	BPOS (10)	BPOS (10)	BPOS (10)	YNEG (9)	YNEG (9)	YNEG (9)	YNEG (9)
MIDDLE_COIL	BPOS (10)	BPOS (10)	YNEG (9)	BPOS (10)	YNEG (9)	YNEG (9)	RPOS (6)	YNEG (9)
BOTTOM_COIL	BPOS (10)	YNEG (9)	YNEG (9)	YNEG (9)	YNEG (9)	RPOS (6)	RPOS (6)	RPOS (6)

Table 2.1: Coil Arrangements For Each Slot

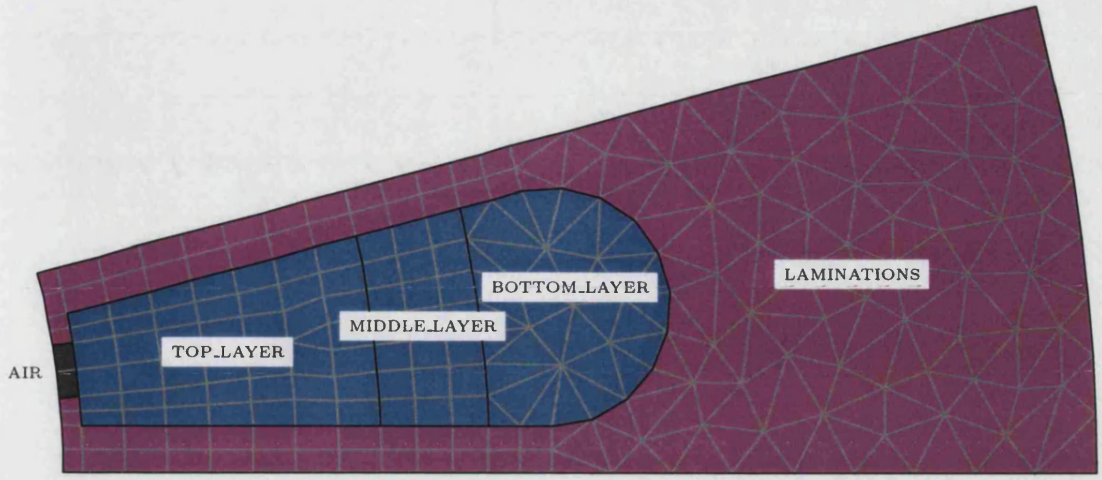


Figure 2.23: All Mesh Tiles Grouped In To Regions

Let us start by creating the region properties for the six coil regions we're going to need. The region editor automatically creates properties for new regions, the *TOP_LAYER*, *MIDDLE_LAYER*, and *BOTTOM_LAYER* regions all use identity 6 so this will be the only new region property. In figure 2.24 this has been edited into region *RPOS* and the remaining five coil regions of table 2.1 have been created in figure 2.25.

This is where we construct a *region mapper* that will ensure that when the component geometrically copies our slot, producing the entire stator, it also incorporates the mapping of regions into this process. Given table 2.1 we can see that given any slot and the name of a region we can return the region property for that region. A single region mapper performs this task and given the number of slots, and regions per slot, we have a lot of arguments to supply; a region map editor has been devised to wrap a more user friendly interface around the process of creation the map.

The region map editor allows the editing of multiple region mappers, just as the region property editor handles editing of all region properties. We will need one map for the stator but may need one for the rotor also.

Each mapper is capable of handling multiple regions. If we are to map the stator slot as one entity we want to encapsulate the mapping of all its regions in one mapper; this allows a simple one to one relationship to be established between

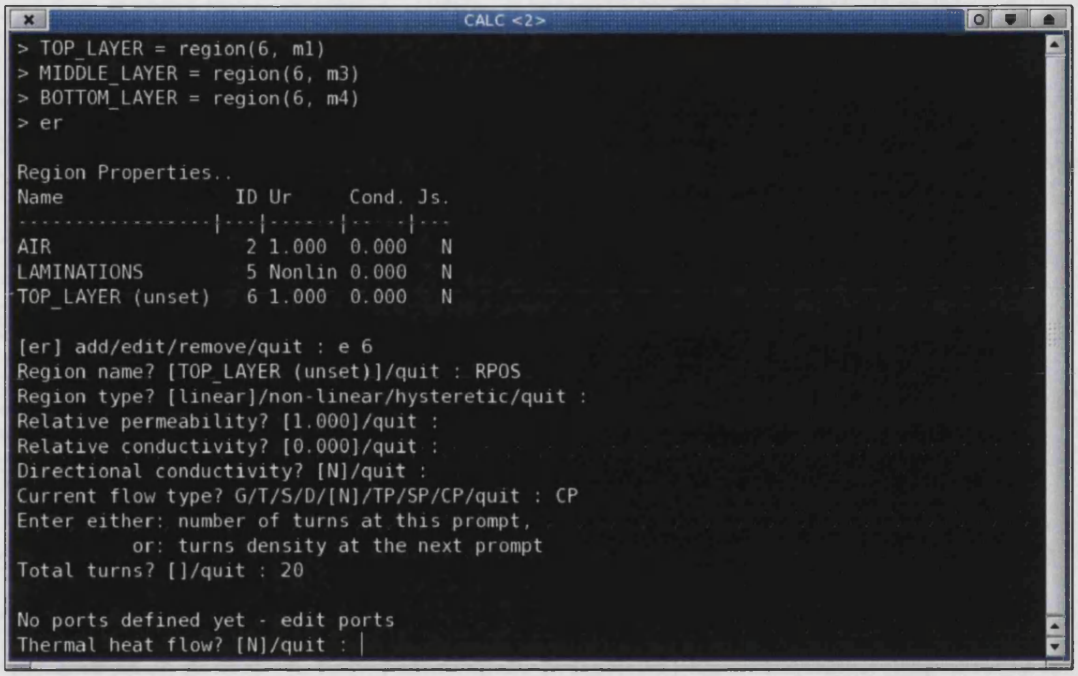


Figure 2.24: Editing The Region Properties Of The Coil Regions

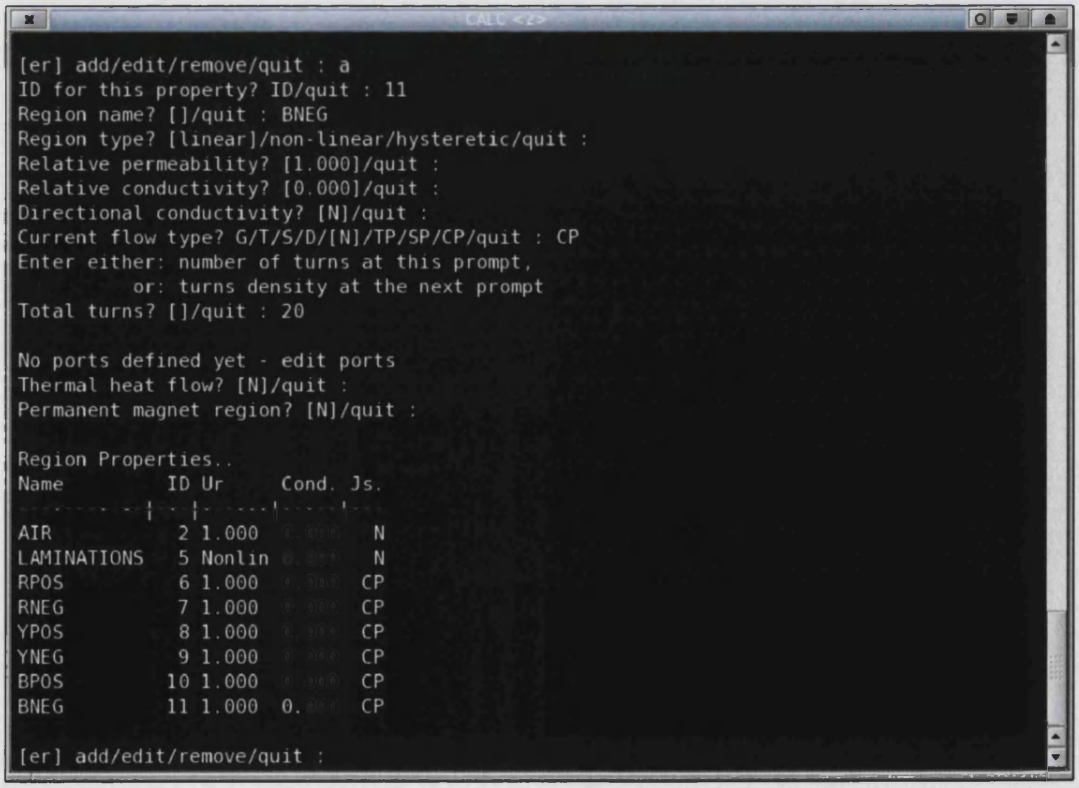


Figure 2.25: Adding The Final Coil Region *BNEG*

the mapping of regions and the mapping of geometry, it is useful when we reuse old designs and an import tool needs to automatically establish this relationship. Figure 2.26 shows the creation of a region mapper and the addition of a mapping for the *TOP_LAYER* region. This type of mapper recreates the layout of table 2.1, it's simple but allows any mapping to be entered however complex the pattern. Simpler patters can utilise simpler region mappers. Figure 2.27 shows the final mapping of all three coil regions, reproducing the entirety of table 2.1.

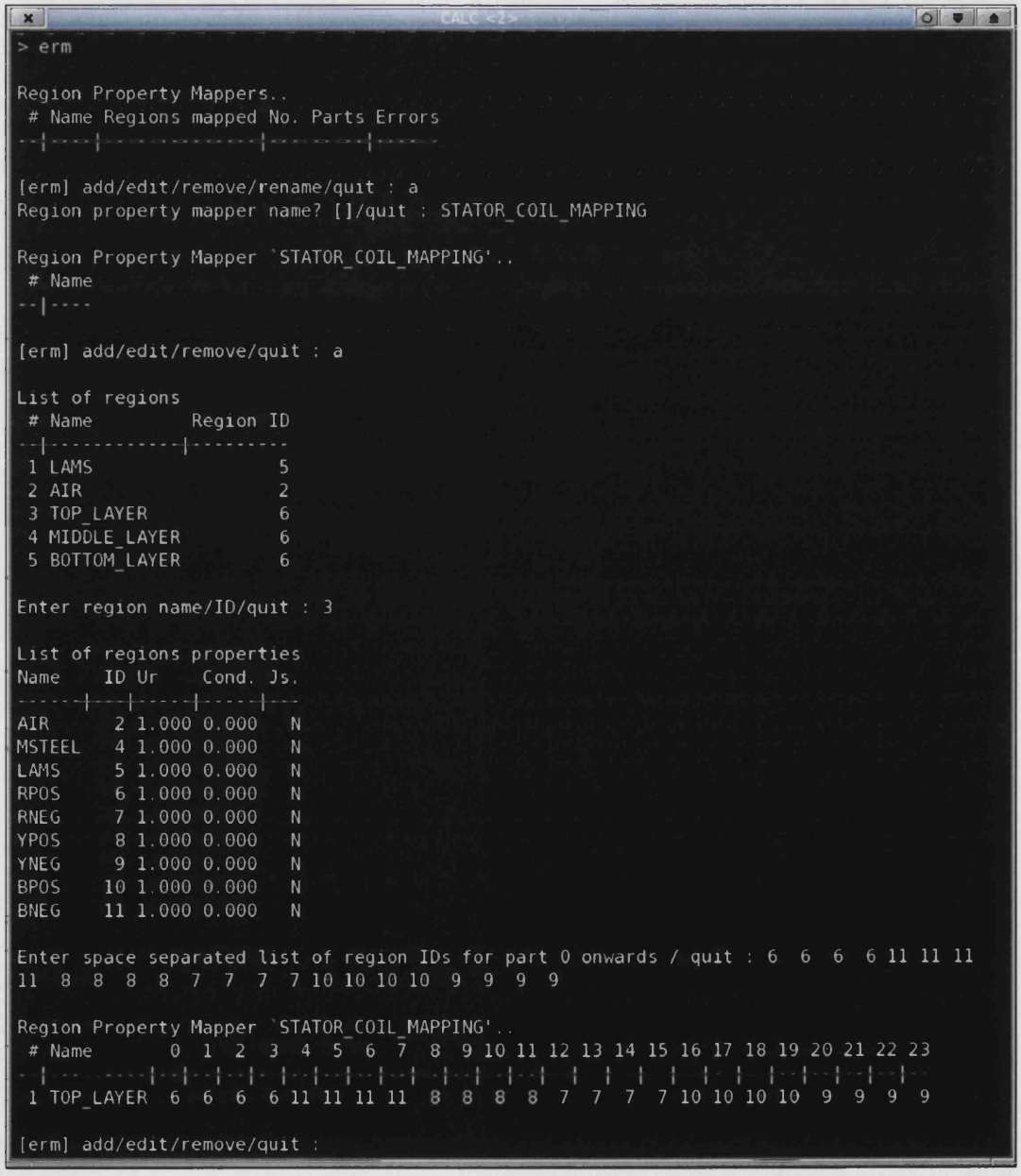


Figure 2.26: Adding Region Mappings For The Stator

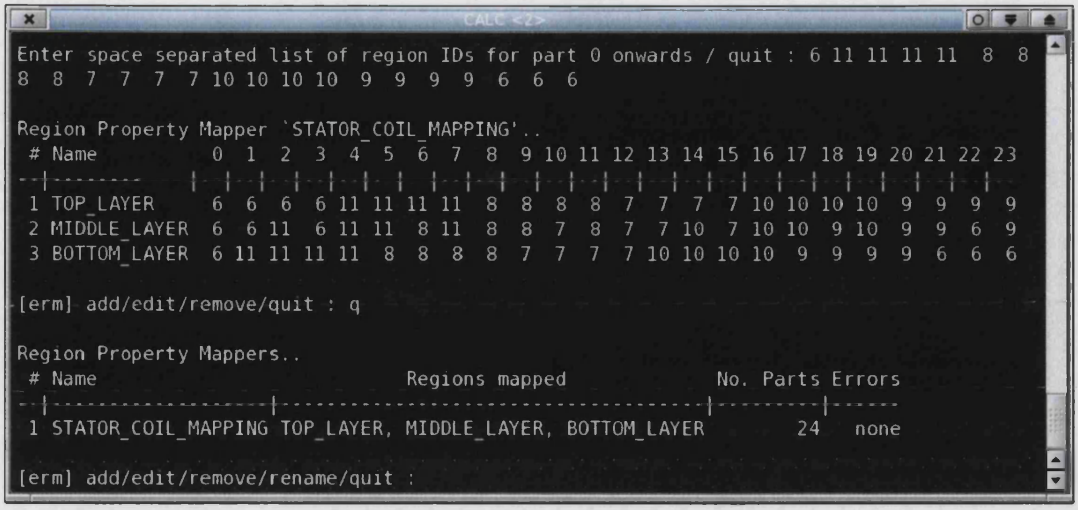


Figure 2.27: Completed Region Mapping For The Stator

2.6 Geometric Mapping

This object oriented design isn't limited to the use of rotating electrical machines. When we copy the geometry we've designed so far we wish to allow any geometric mapping, to cope with linear machines as well as rotating for instance. A geometric mapping object simply maps a coordinate in space, the *rotate* mapper rotates along any axis around a centre. The following mapping will rotate around the centre of our stator, on the z-axis, through the angle *a2* degrees occupied by our slot:

```
geomap = rotate(ctr, 0, 0, a2)
```

2.7 Components

Finally we come to the most complex of our electrical machine building blocks, the *component*. A component takes a list of regions and puts these regions together to construct a single part of an electrical machine. The component looks at the regions' representation of interfaces and elements and using the geometric mapper it constructs a copy of the original regions that have been geometrically mapped in space. It creates new interfaces and elements required for this mapped part.

The component repeats this mapping process if required, it therefore needs to know how many copies of the initial part are required. For our twenty four slot machine we will group all the slot's regions as a single part and tell the component to copy this twenty three times.

Once the component has completed the geometric mapping process it looks again at the interfaces it has created. During the mapping process it took care not to duplicate interfaces. For instance the first copy of our slot, figure 2.28, allows reuse of the interface between the adjacent slots. Duplication of interfaces would result in duplication of nodes. Once mapping is complete many of the interfaces will now be internal, the interface between the two slots of figure 2.28 for example. Internal interfaces are ignored, the component, like the region, is only interested in external interfaces. For each external interface the component creates a discrete segment should one not exist, the original regions are already defined by discrete segments. The discrete segment allows access to the external interface in order to set boundary conditions, periodic boundaries and sliding interfaces.

The component also creates regions that reside in the dependency tree, these correspond to each region mapping and like the original regions they are responsible for propagating region identities through to elements. The component uses the region mapper to determine the region identity of each mapped region. Figure 2.29 shows a section of the final stator with its mapped regions.

We now have a complete stator. Had our winding arrangement been a little simpler, more periodic, we could have told the component to repeat enough parts as to form half or quarter of the stator. The external interfaces the component created would have allowed us to set periodic boundaries and we wouldn't have to model the entire machine.

2.8 Exporting Components To Augment Libraries Of Reusable Parts

Before we complete our electrical machine we will demonstrate how this component can be utilised in future designs through the export and import of components. Then we will import the rotor to our electrical machine to complete this

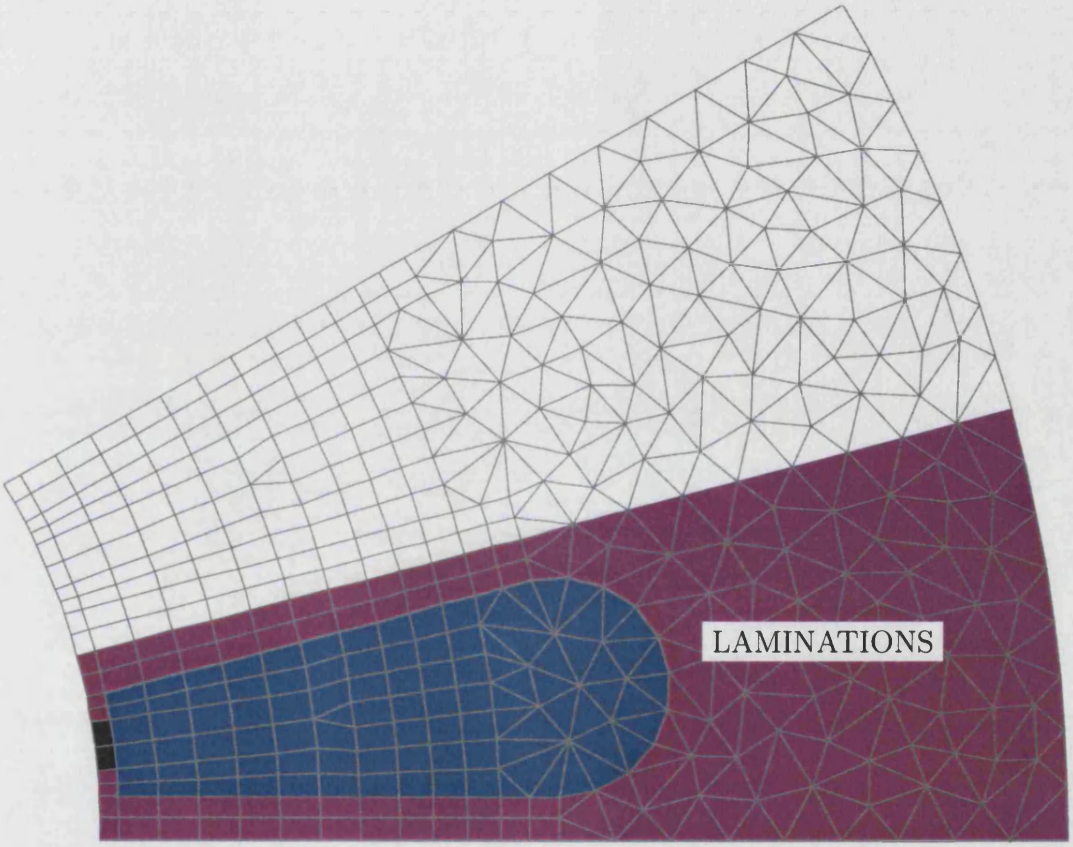


Figure 2.28: Component Groups Regions Into A Part Of A Machine, Ready For Mapping Through To Successive Parts

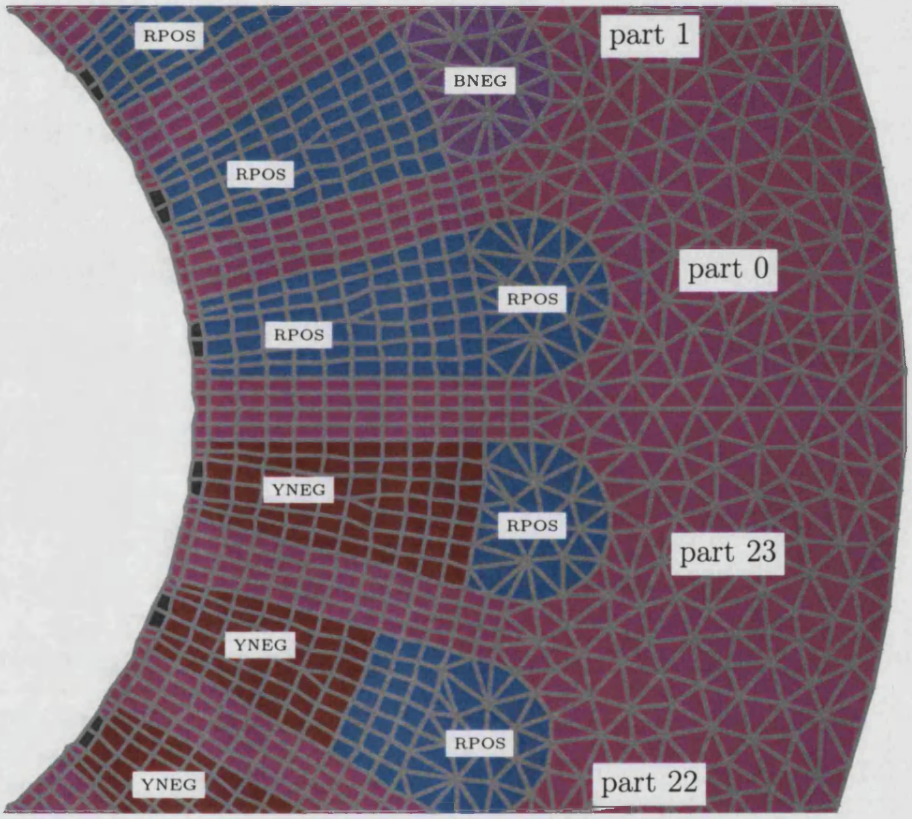


Figure 2.29: Region Mapping Of The Stator

design.

Exporting works with components through the examination of the dependency tree. Every building block of the machine used by the component is linked to it through this tree, by exporting the component we are saving everything needed to reuse this part in a future design. This includes the regions and their mapping, nothing is thrown away. The import process ensures that none of these building blocks can interfere with an existing design, importing a rotor can't interfere with our stator, so keeping as much as possible could later minimise the amount of work we do. The fact that our geometry is governed by a few fundamental parameters allows it to be imported and slotted into place in any future design.

The export process examines our component to ascertain whether its part of a rotating or linear machine. The export process isn't limited to these parts but if it recognises the part it can perform some sanity checks, mainly to ensure the correct parameters exist for it to be imported properly at a later stage. Figure 2.30 illustrates the export process. Part of a rotating machine has been identified and the required parameters are present. Additional parameters have also been identified, these are values that sit at the top of the dependency tree. These parameters can be optionally highlighted in the exported component and given verbose descriptions. When this component is imported the optional parameters will be highlighted so the designer is aware of any design specific parameterisations.

2.9 Importing Library Components

A library component differs from a saved model only in the extra information it has regarding the parameters it uses to slot into a design. The import tool detects this information and presents the designer with a few questions that will be used to set these parameters. In figure 2.31 we are importing the rotor for our machine, part of a rotating machine has been detected and a few questions are presented so the internal diameter, external diameter, and the number of slots can be set. This is all the information we need to fit the import into our design.

From the number of slots we can set the angle of the slot *a2*, the absolute angle of

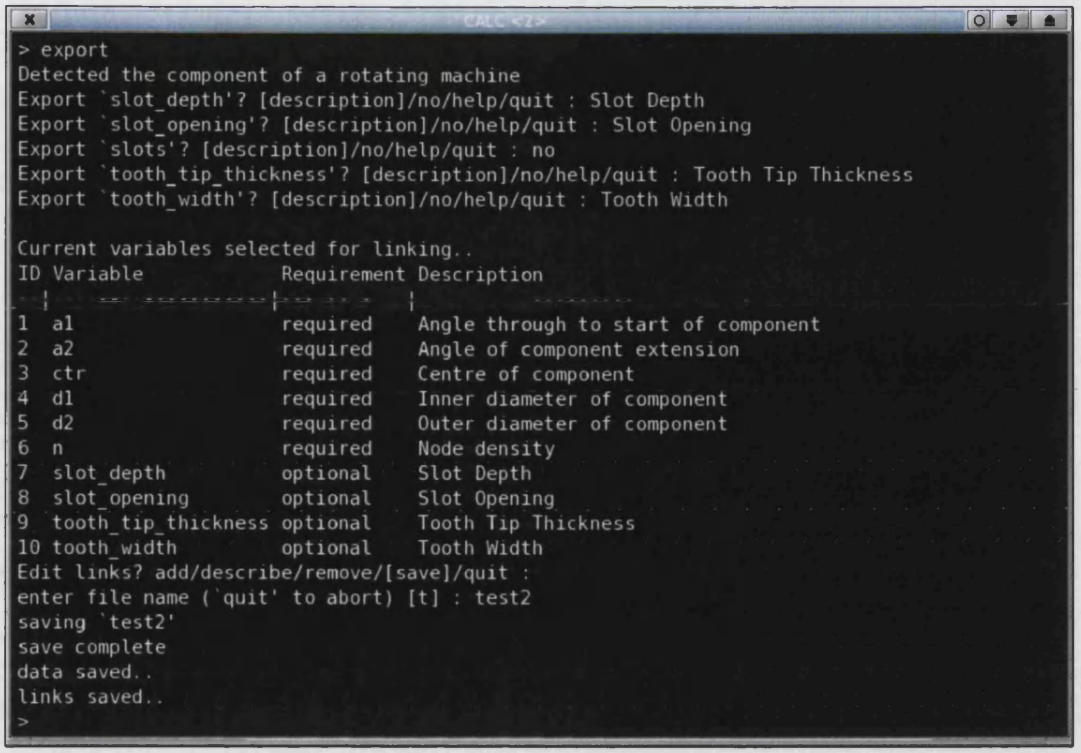


Figure 2.30: Exporting A Component To Augment A Library Of Reusable Parts

the slot *a1* is assumed to be zero and the centre is assumed to be the origin. The import tool can now fix all the required parameterisations of the imported part and construct a component that will form, in this case, the rotor of our machine:

```
d1 = <set from input>
d2 = <set from input>
slots.rotor = <set from input>

ctr.rotor = vvpoint(0, 0)
a1.rotor = 0
a2.rotor = 360 / slots.rotor

repetitions.rotor = slots.rotor - 1
geometric_map.rotor = rotate(ctr.rotor, 0, 0, a2.rotor)

region_map.rotor = <detected from library>
```

```
component.rotor = component(geometric_map.rotor,
                            repetitions.rotor,
                            region_map.rotor,
                            <plus all regions exported>)
```

The import tool asks for a label, a name-space, that will be appended to each object's name upon import. This ensures that names don't clash and the import of one part doesn't affect another part already in our model. The import uses the region mapper of the exported part, or an empty one otherwise, when constructing the component. The one to one relationship between a component and region mapper is extremely useful for the import process; it can construct a component with a single mapper that the designer can edit using the editing tool. No matter how many regions are added or removed from the mapper, as it's still a single object the designer never has to touch the command line and redefine the component in terms of other mappers.

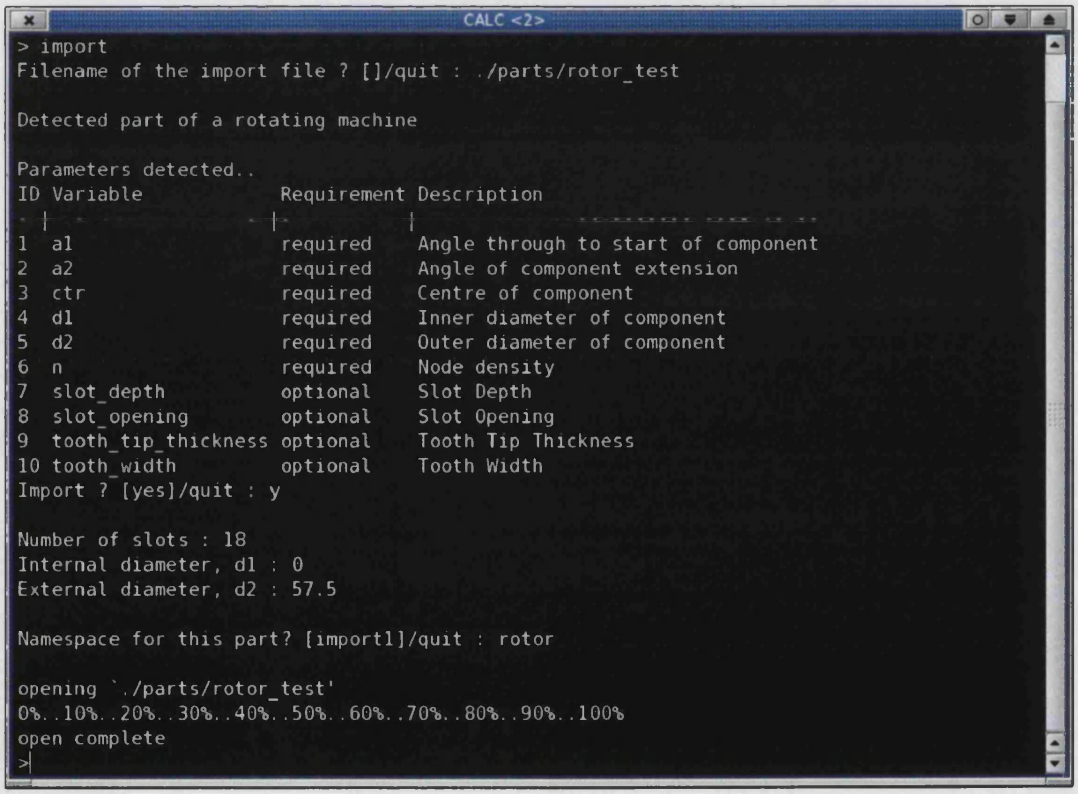


Figure 2.31: Importing A Library Component

2.9.1 Defining Interfaces Between Independent Meshes

We now have two completely separate meshes, one for the stator and one for the rotor. The internal diameter of the stator's mesh matches the external diameter of the rotor's mesh and so their interfaces overlap. However it is unlikely that the nodes along these interfaces will be aligned, we would have to match the geometry and node density on the respective discrete segments and this would significantly complicate the simple import and reuse of components at a later stage.

The chosen solution to this problem is to use a sliding interface[14] which is a fairly general technique that allows independent meshes to be translated and rotated whilst coupled together using Lagrange multipliers. The meshes of our stator and rotor are now free to move with respect to each other without any need for remeshing.

This technique must be implemented within the finite element solver, here we just mark the respective interfaces so the information can be propagated through to the exported mesh.

If we stop the components from mapping parts, by setting the number of times they copy to zero, then we'll highlight the discrete segments that are forming the sliding interface in figure 2.32.

```
repetitions = 0
repetitions.rotor = 0
```

Manually we would mark these discrete segments as sliding interfaces as follows:

```
> list ds15.rotor
ds15.rotor=dsegment(cc1.rotor, _pe_cl1.rotor, _pe_cl3.rotor, _vn_ds15.rotor)
        segment: cc1.rotor
        from   : _pe_cl1.rotor
        to     : _pe_cl3.rotor
        with   : 15 nodes
```

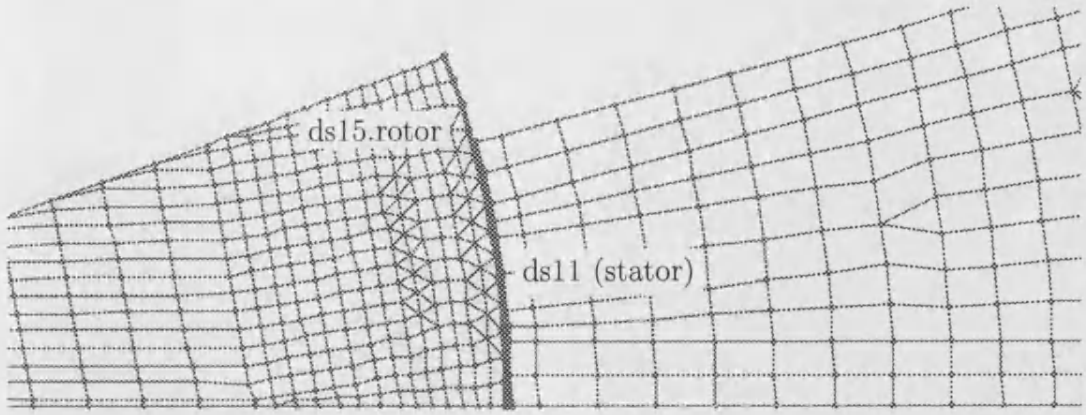


Figure 2.32: Identifying Discrete Segments That Form A Sliding Interface

```
> ds15.rotor=dsegment(cc1.rotor,
                      _pe_cl1.rotor,
                      _pe_cl3.rotor,
                      _vn_ds15.rotor, flag(s))
```

The discrete segment takes an optional argument that can be used to change its properties. When the component copies parts and creates new interfaces the properties of the interfaces it copies are propagated through the copies. It is possible to set the attributes of the whole of the stator's internal interface, and the rotor's outer interface, by designating two discrete segments as sliding interfaces. The, currently rather primitive, interface editing tool of figure 2.33 uses this feature and presents its user with the short list of discrete interfaces that it finds external. As their properties are propagated, they may only represent a small part of the whole interface but they actually control all of it.

2.10 Using The Mesh

Here are some gratuitous screen shots that show the export of the mesh to a format that can be read by the MEGA[15] finite element solver, figure 2.34.

A view of the mesh via the MEGA pre-processor is in figure 2.35.

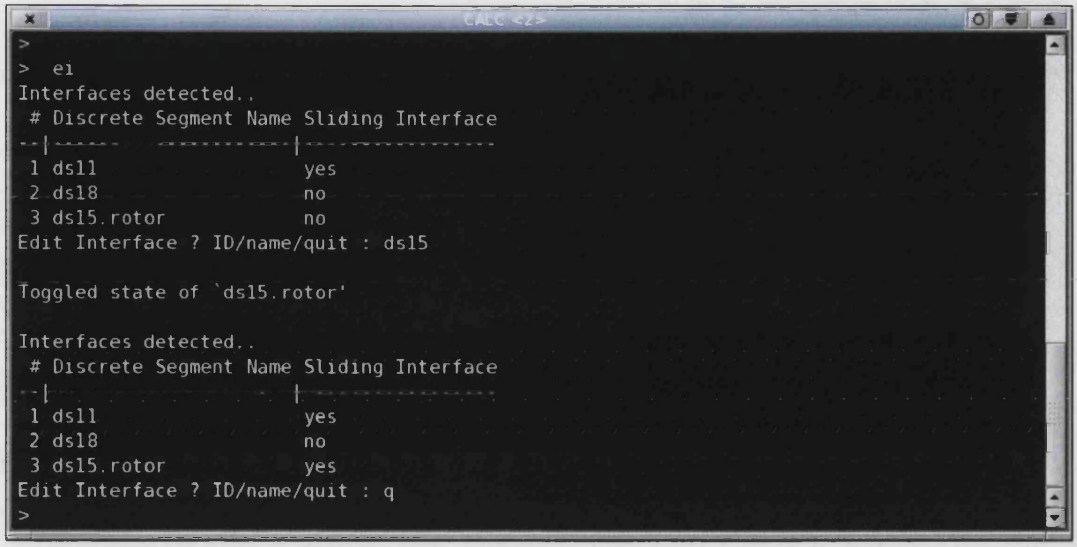


Figure 2.33: Toggling The State Of Sliding Interfaces On Discrete Segments

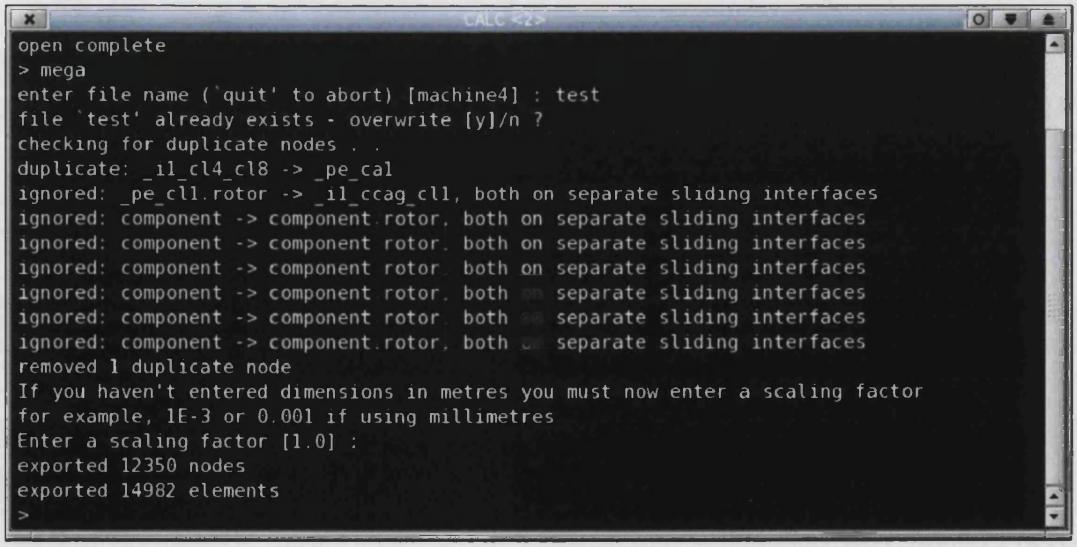


Figure 2.34: Exporting Mesh To A Finite Element Solver

Figure 2.35: Exporting Nodes And Elements To A Finite Element Solver

2.11 Transformation Of The High Speed Motor Into An Induction Motor

If we wanted to use a different rotor configuration, to turn this into an induction motor, we can put our reuse ability to the test with the import of another library component. Figure 2.36 shows the geometry of an induction machine's rotor, a definite case of here's one I did earlier. Figure 2.37 gives the more attractive and colourful illustration that better identifies the rotor's regions.

Finally, in figure 2.38 we can see full induction motor using MEGA's pre-processor. End of gratuitous screen shots.

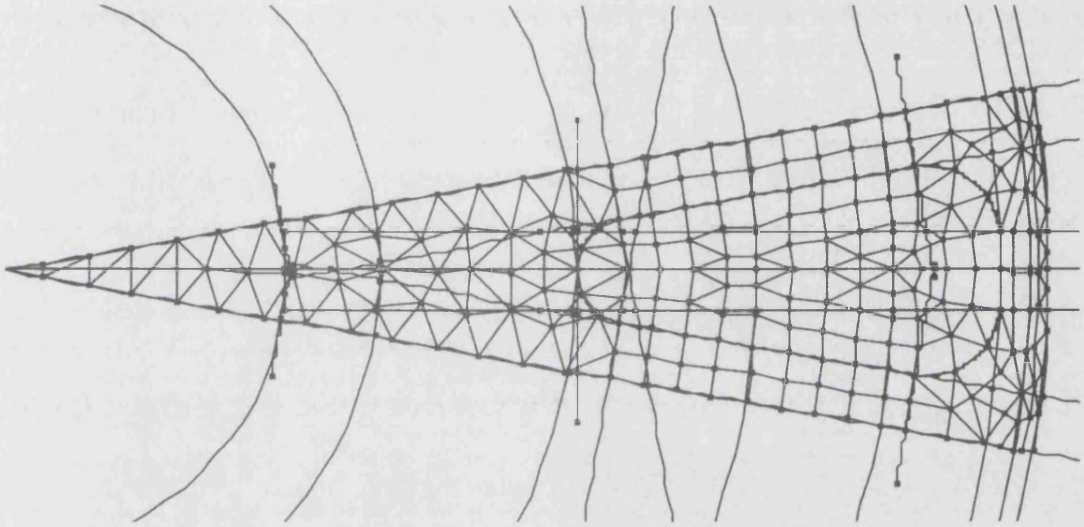


Figure 2.36: Rotor Library Component Of An Induction Machine

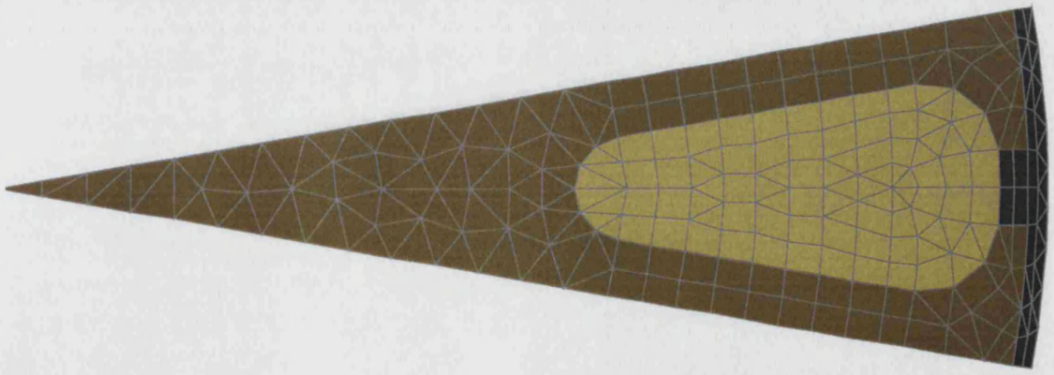


Figure 2.37: Rotor Library Component Of An Induction Machine With Regions Highlighted

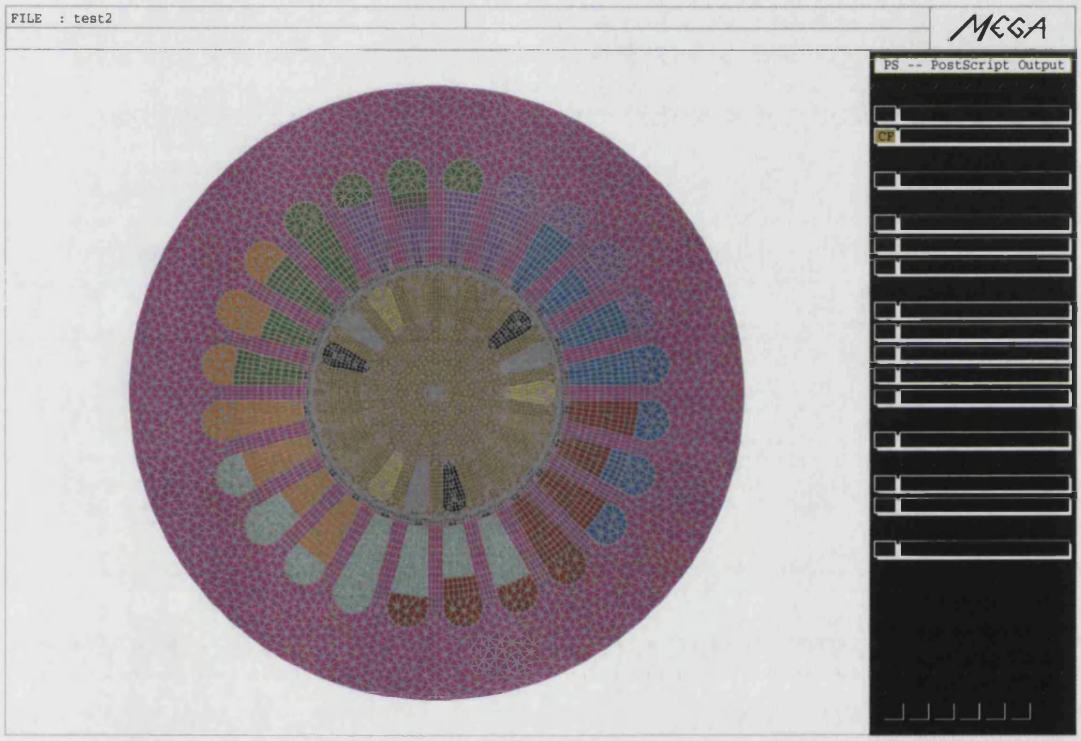


Figure 2.38: Rotor Of Induction Motor Imported

Chapter 3

Communication Between Objects

To make a point communicate with a line when its position changes, so that the line can change accordingly, we need a form of communication between our data objects. This communication should work across all objects without the need for any unique implementation beyond basic necessity, after all we are using an Object Oriented Programming scheme.

3.1 Parents and Children

The design approach used starts with basic numeric values, independent of other data objects, working up to symmetry components. This relationship is the basis of a dependency tree, once a value changes the dependent objects need either to be updated immediately or flagged to ensure an update occurs when they're actually used. The dependency works in one direction through the tree. The initial values are parents to the points and other objects that ultimately end with the components. We can consider the dependency in terms of parents and children. The initial values are without parents, subsequent objects, taking these values as arguments, are the children of these values; these objects in turn become the parents to subsequent objects that take these as arguments in their definition. The dependency therefore involves the use of a list of parents and children within an object. Creating this as a stand alone object allows reusability, objects inheriting this dependency object can all be referenced as a dependency object regardless of

their final form. Our dependency tree becomes a network of dependency objects, each of which contains a list referencing the parent dependency objects and child dependency objects. A polymorphic function within the dependency object need only be implemented by the inheriting object, the purpose of which tells that object to do the necessary update to its state.

The abstract nature of the dependency object has resulted in it being called a *Thing*. An example illustrating the dependency relationship is as follows, starting with the creation of two values:

```
x = 1
y = 2
```

These values will always be without parents and are childless until we define a cartesian point referencing them as attributes:

```
p = vvpoint (x, y)
```

Now both the values have the point as a child, the point has the two values as its parents. As values and points inherit the dependency object, *Thing*, the lists of parents and children will point to the dependency object part of the value and point object's structures. Hence it is possible to reference any *Thing* derived object in these lists. This poses the question, once an object is referenced in this list then how do we tell what type of object it is? It could be a value, it could be a point. To the outside world, looking at the object, some kind of mechanism would indeed be needed to establish the type of object once its identity is lost and it becomes a generic *Thing* reference in one of these lists. Something easily remedied with run time type identification, a mechanism found in the C++ OOP used, or the use of text fields identifying types and families. However the point, being a concrete representation, knows it was constructed from two values. The two references in its parent list must thus identify these two values. From this we can deduce that the order of the parents, contained in the parent list, must never be tampered with.

If we were to add a third value and a second point as follows:

```
x2 = 2
p2 = vvpoint (x2, y)
```

The y value now has two points as children. From this it can seen that objects will always have a fixed number of parents, the order of which must be maintained, however the child lists can contain an arbitrary number of references. If we were to delete point p such that its reference were removed from the child lists of values x and y, it can be seen that the order of the child lists is unimportant.

3.2 Public Accessibility

To facilitate the explanation of dependency operation, this is how the basic *Thing* looks, table 3.1 shows the functions that are provided publically, to the whole world, and table 3.2 shows the protected functions that are accessible only to derived objects.

The purpose of public functions are as follows:

family, type :

A means of identifying an object's family and type. Families of objects allow interchange of family members within the dependency tree, chapter 5 explains this further. The earlier example include objects of the *value* and *point* families. The type identifies a particular object, all types being unique. The example included the *vvpoint* type of the *point* family in our example.

label, name, namespace :

Each dependency object has a unique name, used to identify it within the model. For the floating point value "x = 5", x would be the assigned label. A name space may be used, such as *width.slot1* and *width.slot2*, allowing like named variables to be used in different areas of the electrical machine. The label is the summation of the name and name space using the full stop as a separator, any number of parts may exist to a name space; this allows the nesting of name spaces, such as *width.slot2.machine*, with the name space

Public Methods	
	Thing ( )
virtual	**~Thing** ( )
virtual const char*	**family** ( ) const = 0
virtual const char*	**type** ( ) const = 0
const char*	**label** ( ) const
const char*	**name** ( ) const
const char*	**name_space** ( ) const
const ThingList&	**parentCList** ( ) const
const ThingList&	**childCList** ( ) const
bool	**flag** (unsigned n) const
void	**setFlag** (unsigned n, bool b)
void	**flagDependents** (unsigned n, bool state)
bool	**pending** ( ) const
bool	**invalid** ( ) const
bool	**retired** ( ) const
bool	**orphaned** ( ) const
virtual bool	**retire** ( )
virtual bool	**reinstate** ( )
bool	**reco** (bool down = false)
virtual bool	**canWrite** ( ) const
virtual void	**outX** (ostream &out)
virtual void	**outG** (ostream &out) const
void	**outFamily** (ostream &out) const

Table 3.1: Thing Public Interface

Protected Methods	
void	**setLabel** (const char *str)
ThingList&	**parentList** ( )
ThingList&	**childList** ( )
void	**adoptParent** (Thing *parent)
void	**disownParent** (Thing *parent)
virtual bool	**_reco** ( )

Table 3.2: Thing Protected Interface

operating in a like manner to internet domain names[16]. Future work, chapter 9 would concentrate on an intuitive graphical user interface that allowed the designer to use multiple windows viewing the same machine, each view allowing confinement to a name space.

parentCList, childCList :

A method of read only access to the list of the object's parents or children, this prevents unwarranted access to this sensitive data allowing miscellaneous access to, for instance, verbosely print dependency information. Write operations on this data is performed through the *parentList* and *childList* functions of the protected interface.

flag, setFlag, flagDependents :

To be versatile and efficient on memory the *Thing* stores boolean flags as the states of different bits within one integer variable, similar to bitfields within C[17], only new flags can be requested, or unused flags relinquished. With this mechanism a new flag can be requested, reserved for use across all dependency objects, for the lifetime of a particular operation. This method is used when saving data, marking objects as saved whilst ensuring no object is written out until all the objects it is dependent upon have been. The *flag* and *setFlag* functions read and write the boolean state of a flag according to an index that identifies the bit held to store this information. In order to mark all the dependents of an object, children and grand children and so on, *flagDependents* is used.

pending, invalid :

Changes need not take take effect immediately, should the state of an object change it can mark any dependent objects as requiring an update by setting their *pending* flag. Only when an object's state is accessed does it need to check the *pending* flag and update its state if necessary. For *invalid* objects, imagine two segments, once intersecting, no longer intersect; the intersection point is now invalid. Invalid objects should not allow their state to be read, the effect could be damaging if not just incorrect. Both these functions provide access to flags using the *flag* function, they simply used stored indexes pointing to the correct flags.

retired, orphaned, retire, reinstate :

More flags are provided by *retired* and *orphaned* using stored indexes; these flags offer an insight into the state of the model precipitated by the *retire*

and *reinstate* functions. These functions are better explained in chapter 6.

reco(bool down = false) :

A very essential function meaning "update your state if necessary". This is the easy way to make sure the object is current and thus whenever an object's state is accessed, this should be called first. The function will recursively access all parents or children, depending on whether the *down* variable is set to *false*, the default, or *true* respectively, updating their state if necessary. This function can therefore be used to either ensure that all parents are up to date, so that this object can be updated, or all children are up to date, because this object has been changed. The value returned by this function is *true* if the object has a valid state, allowing the object to be ensured updated whilst accessing its validity simultaneously, and thus whether to use the object. The pending flag is used by this function during the recursive check of parents or children to determine whether anything actually needs to be done; hence efficiency is maintained by avoiding updates of current states. This function is further detailed, later in this chapter..

canWrite, outX, outG, outFamily :

All objects can output a list of their dependencies in the form of "name = (parent1, parent2, ..)", this is done with *outG* and provides the format used for saving the model to file. However, further information on the objects state may be provided if the object implements *outX* and *canWrite*. Values do this in the *GValue* base object to print the real and complex parts of their representation. Finally, *outFamily* will print a nicely tabulated display of the parents and children of an object. This provides knowledge of the object connections and has been a very handy tool in the debugging of dependency tree manipulation tools.

The *Thing's* protected functions are accessible only to objects inheriting from *Thing*. This interface gives a little more access to the object's representation:

setLabel :

Allows the object's unique name to be set.

parentList, childList :

Methods providing read and write access to the list of the object's parents

and children.

adoptParent, disownParent :

Usually *adoptParent* is called when a concrete object is constructed, this function attaches the calling object to a parent such that it becomes the parent's child. This object has then become part of the dependency tree of the model. The reverse action to this, *disownParent*, detaches this object from the stated parent, isolating it and any dependent objects from the parent and most probably the dependency tree.

_reco :

this polymorphic function is to be implemented by the concrete object representation ultimately inheriting this functionality. The function implements the concrete object's action of reading in data from parents in order to update its state; it implements the mechanism of converting the data acquired from the parents, defining its interface, into the generic representation that is usually defined by the base class object of its family. The function is only called from the generic *reco* function and returns a logic, boolean, value indicating the success of the conversion; if false, the object becomes *invalid.*

3.3 Reconstruction of Dependency Tree Members

One function within *Thing* is of great significance, explaining the *reco* function gives a good insight into the operation of the dependency tree. When the data of an object is accessed, through its interface, *reco* should be called to ensure the state of the object is current and valid. This is usually automatic, for instance the display of objects will call this function and skip display of the object if it is invalid. The display function returns no data, however functions returning data on this object should take this measure to ensure validity of the object before accessing its data; if the object has an invalid parent, and is thus invalid itself, the action of reading invalid data could be damaging, the object may be without data being unable to read the necessary parameters defining it from the parent. Most data access is done as a result of updating the dependency tree, this action

itself safeguards against this eventuality through the *reco* function and as a result the use of *reco* is seldom used externally.

Here begins the definition of *Thing's reco* function:

```
bool
Thing::reco(bool down)
{
  if (!pending()) return !invalid();
  Thing *parent;
  ThingListIter parents(_parentList);
  while ((parent = parents.next())) {
    if (parent->pending()) if (!parent->reco()) return false;
  }
  if (!_reco()) {
    setFlag(PENDING, false);
    setFlag(INVALID, true);
    flagDependents(PENDING, false);
    flagDependents(INVALID, true);
    return false;
  }
  setFlag(PENDING, false);
  setFlag(INVALID, false);
  if (down) {
    Thing *child;
    ThingListIter children(_childList);
    while ((child = children.next())) child->reco(true);
  }
  return true;
}
```

Now we'll explain each significant block in turn, starting with the opening of the definition. By default the boolean value of *down* is *false*, meaning update parents only when calling this function. The mode of operation can be changed by inverting this value, resulting in changes immediately cascading down the dependency tree once this function is called.

```
bool
Thing::reco(bool down)
{
```

When a parent changes state it flags all children as pending, if our pending flag is not set then nothing has changed. Return immediately with the value of *!invalid*, i.e. logical true if we're valid.

```
  if (!pending()) return !invalid();
```

Getting this far means we're *pending*, we need to ensure the state of our parents is valid before checking our own state. The following cycles through all parents checking validity. Bearing in mind that this prompts the same action in our parent that we're currently stepping through ourselves, then if one parent should prove invalid we stop immediately. That parent will see that it marks itself invalid along with all dependents, that's us, so we simply return *false* stating that we're in an invalid state.

```
  Thing *parent;
  ThingListIter parents(_parentList);
  while ((parent = parents.next())) {
    if (parent->pending()) if (!parent->reco()) return false;
  }
```

The above has certified our parents valid and their state current. We call the polymorphic function *_reco* to get the concrete representation to convert its defining data into the generic representation. This function will return *true* if successful, *false* otherwise. In the unsuccessful case we're no longer pending and we're no longer valid too. We can mark all dependents likewise to ensure the program doesn't go and extensively check them too. Should they have called our *reco* function, they would abort at the above stage. The idea is to minimise the processing involved.

```
  if (!_reco()) {
```

```
    setFlag(PENDING, false);
    setFlag(INVALID, true);
    flagDependents(PENDING, false);
    flagDependents(INVALID, true);
    return false;
  }
```

With the last stage proving successful we can mark ourself as current, not pending, and valid.

```
  setFlag(PENDING, false);
  setFlag(INVALID, false);
```

The exceptional mode of use, now we ensure all our children update their state. They will perform the above; they will stop at the first, parent non-pending, stage for the case where the parent is us, and will recursively check upwards for any other parents. Therefore, they may still prove invalid, even though we are valid, due to one of the other parents.

```
  if (down) {
    Thing *child;
    ThingListIter children(_childList);
    while ((child = children.next())) child->reco(true);
  }
```

All is successful, return a valid status of *true*

```
  return true;
}
```

For what use is the exceptional mode when our dependency oriented data structure does not use this method? Object Oriented Programming is, among many things, designed to make reusable code by producing objects dependent on as few other objects as possible. The application of these objects to other areas

saves time in the future. Not wishing to bind the graphical user interface of this program to a particular graphical system, the used method of graphical widgets were abstracted behind my own interface. The idea being that should another graphical system be used, that implementation could be made behind my abstract interface and the new system plugged in as a replacement. One difficulty in this is the placement of graphical widgets within the physical window, they do depend on one another for their placement. Therefore, the widgets were inherited from the dependency object. Re-using the *InfixGValue* to provide a means of describing the placement of widgets, the object builder of chapter 5 was reused to construct widgets along with value objects that depended on and controlled the x, y, width, and height values of the widgets. The "downwards" action of the dependency tree is used, for example, such that if the window size is changed, that change immediately cascades down the dependency tree updating dependent widgets; the effect of the re-size being immediate, rather than postponed as our model data structure facilitates where changes need only take effect when needed.

3.4 Constraints

Dependency works down the tree from the real and integer values to higher level objects. This motion simplifies the inter-object relationships as an object will always be constrained by its parent. Different concrete representations, chapters 2 and 7, allow the same types of object to defined in terms of different parameters which become their parents. Arcs, lines, and circles are all segments, concrete representations of these segment types allow them to be created in any manner necessary for a design strategy since the addition of a concrete representation requires no modification to the existing objects. Amongst the arc's concrete types defined exist contrasting variations, one specified by its start, through and end point, another using a center point, start point and end point, a third using a center, radius and angle values. Differences between these lie in the order of dependency; a three point arc depends on the position of the three points, a center, radius arc depends on the center point and the other values; this type creates points at it's start and end which depend on itself, this is the default behaviour for objects that are not defined by points as they themselves construct points dependent on their open ends to facilitate connection to other geometries, see chapter 6. Therefore, the two types explained differ significantly

in their constraining behaviour. The dependency tree governing the model filters changes down the tree so behaviour is dictated by dependency. Rather than needing explicit constraints, the dependency relationship implicitly defines these; a line dependent on two points must vary it's length according to the distance between points, whereas a line defined by a starting point, angle and length will create end points which must move as the line's length changes. This constraining effect simplifies the desired parameterisation response within a model and it's effects are immediate. It avoids the need to explicitly define constraints between all objects in order to effect the desired parameterisation. The cost of iterating over simultaneous equations resolving these constraints is also avoided. A state change affects dependent objects only, the repercussions echoed in a single pass down the tree, as opposed to iterations of an order corresponding to the number of couplings between objects[13].

Chapter 4

Building Objects, Pre-processing

There are a number of objects related to the mechanism of constructing building block objects and placing them into the dependency tree. The first step involves the pre-processing of user input. Input is taken in the form of a string of characters. Build of an object can always be recognised by a valid object name followed by an assignment; valid object names begin with letters, or underscores if internally generated by the program, thereafter containing any combinations of numbers, letters, underscores and full stops. Names beginning with numbers are specifically disallowed.

Once the character string input is identified, conversion to an *Expression* can commence. The *Expression* breaks the string of characters down into sections according to specific formatting. Expression types exist for different formatting requirements, the mathematical expression handles the assignment format used for building objects whilst the command expression handles other requirements. A mathematical expression breaks the string down so the builder, after pre-processing the expression to expand any shortcuts, is able to immediately identify the variable name, type of object and the arguments supplied to that object. The builder takes the object type and looks for a matching *Prototype.* For every building block object a *prototype* exists of the same name, it's their job to check that the supplied arguments match the object's requirements. They can perform any necessary processing on the argument data, checking types and quantities, before building and returning the object to the builder. A mechanism for type checking exists such that should the supplied arguments specify

non-existing objects, that information can be used to try and build an object of the correct type. This recursive mechanism allows the user to build objects, specifying arguments in terms of unbuilt objects that will be automatically manufactured prior to the main build. Thus it allows nesting of expressions, with it being possible to construct an entire model on a single line of input.

With a completed build, the object is inserted into the dependency tree at the appropriate positions; if the named object already existed in the dependency tree, that object is pulled out of the tree, the new object is inserted and the dependency tree is appropriately modified.

Finally, post-processing is performed by the builder. This may involve the build of subsequent objects specified by the object just built, the re-intersection of affected segments and components, and an update of displayable objects with the display.

4.1 Formatting Input To Expressions

Data entry is either from file, console or graphical user interface. Regardless of the method, data entry begins with a common character stream representation. The stream is checked for the assignment pattern that tells the program an object type is being assigned to a variable name. The *Builder* object takes its input as an *Expression* describing the object to be built.

4.1.1 Expressions

An *Expression* is a linked list of character strings designed to ease the input of user provided text. For that reason, the expression has a few safety features that ensure that, should the amount of input fall short of that expected, empty text strings are always returned when data has been exhausted. This prevents extensive testing throughout the program for the null pointers usually returned by the linked list, of C++ origin, that is reused extensively throughout this program. Instead, the expression has inherited the linked list and extended its interface to provide methods through which its list can be accessed in a protected

manner. As a result, a function operating on a string will at worst operate on an empty string and never a null pointer; that would certainly cause the program to terminate.

The user input, having being recognised as data input, is now converted into an expression by an *Expression Parser*. This involves splitting the stream of characters into individual characters or blocks of characters. It is akin to taking a sentence and splitting the sentence into its separate words by cutting at, and removing, the spaces. The expression would by a linked list of words. Reading the expression with an iterator would result in sequential access of each word, from start to finish, of the sentence. The end of the sentence being denoted by an empty character string. The safety mechanism of the expression would ensure that should another word be asked for, empty strings would thereafter by provided. This protects the program should only one word be contained within an expression where the program expected more.

4.1.2 Expression Parsers

A stream of characters, an istream in C++, provides the input for the *Expression Parser*. As long as input is supplied, the expression parser will process the stream of characters deciding where to cut and store lengths of characters as an expression. Multiple expressions can be entered simultaneously allowing the parsing of files, expressions denoted by separate lines with the line feed separator, and single line inputs, the semi-colon as a separator. Several expression parsers exist, all derived from the generic expression parser that provides the core functionality. A *Mathematical Expression Parser* cuts the input stream separating names, numbers, grammatical symbols and mathematical symbols, passing text within quotes untouched and otherwise stripping all spaces and tabs, white space, from the input. To do this, the parser requires knowledge of what constitutes a name or a number. This knowledge is required else where and therefore a separate object provides this mechanism allowing object reuse. The parser is therefore simplified, easier to maintain, having a few mechanisms to handle quoted text and reversing of the input stream to amend past decisions.

4.1.3 Expression Identifier

The *Expression Identifier* has the knowledge needed to identify strings of characters. It identifies a number as potentially having a decimal point, an exponent *e*, and a unary sign, but disallows multiple instances of these characters. Valid names are identified, allowing alphanumeric characters, full stops and underscores. A valid name is *a2_slot1.namespace*, whilst *2a_slot1.namespace*, actually a shortcut for *2 * 2_slot1.namespace*, would fail. Signed floating point, real, numbers, integers and names can be identified. Mathematical expressions, checking for balanced parentheses, unary, and binary operators, are also identified. This allows short cuts in the input, with correct identification this can be accepted and expanded to the full format recognised by the program. If the format of names were to be expanded, as was the case when name spaces were added, an update to this object would reflect the change across the whole program.

4.1.4 Mathematical Expression Parser

The following are examples of the way an input stream of characters is split, with each block of split characters being stored as the next token of an expression. Spaces denote the splitting of tokens within the expression:

```
# the raw input:
a=ppline(vvpoint(0,0),vvpoint(5,5))
# the expression:
a = ppline ( vvpoint ( 0 , 0 ) , vvpoint ( 5 , 5 ) )

# the raw input:
a=vvpoint(name1a*+2/(3name_b),-y_val)
# the expression:
a = vvpoint ( name1a * +2 / ( 3 name_b ) , -y_val )
# the shortcuts implied:
a = vvpoint ( infix ( name1a * +2 / ( 3 * name_b ) ) , neg ( y_val ) )

b=-b
b = -b
```

```
b = neg ( -b )

b=-3.1e7--b
b = -3.1e7 - -b
b = infix ( -3.1e7 - -b )

b=3(x--2y)
b = 3 ( x - -2 y )
b = 3 * ( x - -2 * y )
```

4.2 The Pre-Build Processor

The *Builder* receives an expression through its *evaluateExpression* interface function. This entry point performs a check on the input, it allows creation of objects with names preceding with an underscore only if an internal flag is set; this allows for file input of *manufactured names* denoted by this underscore. Manufactured names denote variables manufactured by the program itself, these being adjustable only if already existing. The reasoning behind this is that the naming system reserves these names for use by the program at the appropriate instance. Once created their state may be altered, however the unique naming allows the program to identify objects and perform some appropriate house keeping. This check aside, the *evaluateExpression* function then calls *processExpression* to do the real work.

4.2.1 Expression Resolver

Having said the *processExpression* does the real work, it actually delegates it to two intensive tasks. The first of these is the matching of the object type to be built. This task's performed by the *GBuilderExpressionResolver* object. First checking to see if the input method indicates a file read is in progress, it decides whether to check briefly or extensively. For instance, if a file were being read the input is likely to be formatted correctly having been written by the program. This allows a brief resolution of the input by reading the object type as the first name after the assignment, "=", sign. This method can of course fail if the file

was modified, then the fall back is to use the default, and very verbose, method used for general input. Parentheses are checked for the balance of opening and closing brackets, should the expression be of the form:

```
x = ( 5 )
```

then the enclosing brackets are removed. This example would then match the first check, using the *ExpressionIdentifier* object we would recognise this as an integer assignment. This expands to:

```
# integer expansion..
name int 5
```

We have removed superfluous formatting, leaving the essential name, type, and defining value. Passing the integer test, the expression resolver returns the name and type as separate entities to the builder; the expression, as the third entity, is left containing all defining values for the type, in other words the arguments to our variable's type. It can be seen that shortcuts in the input allow, for certain object types, omission of the type of object being created. The second test, should the integer test fail, checks for real numbers. Qualification of the following input would result in the type being set to *real*:

```
# real assignment examples..
name = 4.4    # shorthand for name = real ( 4.4 )
name = 2.3e5  #               name = real ( 2.3e5 )
name = -3.2                   name = real ( -3.2 )
```

Now for a few more interesting types, starting with the negation of objects of the value type:

```
# where y is a variable of the value type,
# x can be the negation of y..
x = -y
# shorthand for..
x = neg ( y )
```

Assignments are another recognised shortcut that don't just apply to values. Taking "a = b", where *b* is a value, this expands to "a = value (b)". Likewise, assignment of points would be performed for "p1 = p2" if *p2* were found to be a point. The pattern "variable1 = variable2" is recognised, *variable2's* type is determined, the type name is found and the expression expanded. How is the type name found? Well, all object types are created using a *prototype*. *Prototypes* will be explained shortly, however they define the construction of an object type. As a result, they can be interrogated to determine the arguments an object, these are the concrete representations, requires. Assignments can be recognised easily because for "variable1 = assignment_type (variable2)", the prototype needed will state that one argument is required, that argument is the same type as the object being created. All that need be performed is a search for prototypes constructing objects of the same type as *variable2* which take one argument, that argument again being of the same object type as *variable2*. This mechanism has an advantage, some object families are harder to clone and the implementation of the assignment object type for that family can be postponed; when it is added, the search through prototypes will pick the new object up and all assignments for that family of objects.

```
# example assignments..
# values..
a = b
a = value ( b )

# points..
p1 = p2
p1 = point ( p2 )

# segments..
line1 = line2
line1 = segment ( line2 )
```

Finally, the last recognised shortcut is for the infix value type used to handle mathematical expressions. The following patterns are currently recognised as valid. Additions to these rules take effect with the modification of the infix identifier within the *ExpressionIdentifier* object. This function simply looks for the patterns of tokens within an expression and is easily modified.

```
# allow 2 a, infix expands it to 2 * a
v = 2 a
# allow 2 (, infix expands it to 2 * (
v = 2 ( x + y )

# deny a (
# these would be confused with nested declarations
l = ppline ( vvpoint ( x , y ) , p2 )

# allow binary operators
v = 1 + x
v = a * y
# allow unary operators
v = 1 + -x  # infix expands to 1 + -1 * x
v = -a * y
```

Once these tests are complete, the only course of action is to assume the type was entered. Then the token of the expression, following the assignment, is examined against the list of all object types that can be built. If a unique match is found, the name and type can be identified to the builder; the remaining tokens of the expression are taken to be arguments supplied for the build of the object. These are now reduced, the parentheses are examined and the expression is compressed for compatibility with nested declarations. For example, a *vvpoint* takes two value arguments, x and y, as follows:

```
p = vvpoint ( x , y )
```

If we were to supply, ignoring the fact that we can use short cuts, $x + 1$ and $y + 1$:

```
p = vvpoint ( infix ( x + 1 ) , infix ( y + 1 ) )
```

We would want to compress this expression as follows, noting that spaces denote separate tokens within the expression, so that the *vvpoint* still received two arguments:

```
p = vvpoint ( infix(x+1) , infix(y+1) )
# the resulting two arguments with all superfluous grammer stripped..
infix(x+1) infix(y+1)
```

This can be performed by compressing the text between the first bracket and comma, this comma and the next comma, and there on until the last comma and bracket. Attention needs to be paid to parentheses. After finding an opening bracket, discovery of a second bracket means we've entered a nested declaration, discovery of a third bracket meaning we've entered further into another nested declaration, and so on. Concentrating only on the arguments for the top level object declaration, we compress all nested declarations by ignoring commas found in nested declarations; this is done by monitoring the number of open brackets, only when we have one open bracket do we honour a comma and separate those arguments into a token.

```
# a fully expanded expression prior to compression..
l = ppline ( vvpoint ( 1 , 1 ) , vvpoint ( 2 , 2 ) )
# the compressed version providing just two arguments for the ppline..
l = ppline ( vvpoint(1,1) , vvpoint(2,2) )
```

4.2.2 Prototypes

With the expression successfully broken down into the name, type and optionally some type arguments, the expression resolver tries to find a matching prototype to build this object.

For every concrete representation of a type of object, such as addition , subtraction and infix value types, there exists a prototype. A *Prototype*, detailed in table 4.1, constructs its associated object. All objects have a *type* and *family* identifier, for a multiplication value the type would be *mult* with the family being *value*. These identifiers are present in the actual concrete representations and their prototypes, we can not only search for a particular type but should we be less sure of a match we can search through family members; this is also useful if we want to replace a particular object with another of the same family.

Public Methods

	ProtoType (GBuilder &builder, const char *type, const char *family)
virtual	**~ProtoType** ( )
const char*	**type** ( ) const
const char*	**family** ( ) const
virtual Thing*	**construct** (const char *name, const Expression &parms) = 0
Thing*	**construct** (const char *name, const ThingList &parents)
virtual unsigned	**parameterTypes** ( ) const
virtual const char*	**parameterType** (unsigned n) const
virtual int	**parameterScope** (unsigned n) const
int	**matchByType** (const char *desc) const
int	**matchByFamily** (const char *fam) const
GBuilder&	**builder** ( )

Protected Methods

	check (const char *myname, const Thing *parent, const char *parentname, const char *parenttype) const
bool	**parameterCheck** (const char *myname, const char *nextparm, const Expression &parms) const

Table 4.1: Prototype Public and Protected Interfaces

Prototypes have knowledge of their respective concrete counterpart required for its build. Upon receipt of an expression containing arguments for the object's build, the prototype will check these arguments for compatibility; should this fail, the prototype will return a null pointer denoting this. In order to preempt this, such that we can be sure of supplying the correct arguments, the prototype has in its interface the functions capable of supplying this information. This is useful, for example, in supplying a graphical user interface for the input of the correct data, or for searching prototypes for specific matches; one such case being the search for assignment types, recognised as requiring one object as a build argument that is also of the same type as the assignment object itself.

The prototype's type specifying interface has three functions, *parameterTypes* tells us how many types of object the prototype requires:

```
# for example,
# the multiplication value requires one type, the value type..
a = mult ( value1 , value2 )

# the pvcircle requires two types, the point and value types..
c = pvcircle ( centre_point , radius_value )
```

The function *parameterType* identifies the required types, *parameterScope* identifies the quantities of the required types. Therefore, if *parameterTypes* told us three types were required, *parameterType* and *parameterScope* could be called three times as *parameterType(0)*, *parameterScopy(0)*, *parameterType(1)*, *parameterScope(1)*, *parameterType(2)*, *parameterScope(2)*. Instead of a specific quantity, *parameterScope* can return *0*, meaning optional, *-1*, one or more required, *-2*, two or more required, and so on.

4.2.3 A Factory of Prototypes

To facilitate in the search for a prototype, *matchByType* and *matchByFamily* functions compare a given, and possibly incomplete, type or family against the respective type or family of the prototype. However this matching needs to be performed for all prototypes, at least until a match is found, and that's where the

factory comes in. A builder, *GBuilder*, has a factory, *GFactory*, and the factory contains a list of prototypes thus describing what it can build. Inheriting from the generic factory, factories can be built with a base list of default prototypes; additional prototypes can then be added later, usually by builders, inheriting the generic *GBuilder*, to create customised factories that produce objects to specific build requirements.

Finding a prototype match via the factory is done so by calling *searchProtoType* with a description of the object, the factory will first try an exact match and then, if specified, try to match as much of the description as possible to likely candidates. Should it match a variable, if that variable exists then parameters for that variable are read from the prototype. At the point of entering a type, other types from the same family are listed to allow the change of the variable within the possible scope. At the point of entering arguments, grammar is examined to determine the argument being entered, taking into account nested declarations, such that a list of available variables, of the correct type, can be listed and completed against any entered text. This applies to the input of new data too; once a new variable name is recognised, all types are listed. When a type is entered, it is checked, the required arguments queried, and possible variable and type completions displayed. This allows the user to enter arguments of existing variables or types for nested declarations; once a nested declaration is entered, completion will act upon the arguments for the nested declaration, to any depth, until it is completed and the higher level argument list completion can continue. This whole process is facilitated by the factory whose *matchByParameters* function will check the argument type list against a prototype, verifying a match, highlighting errors and predicting the types required for completion.

Chapter 5

Building

Having had a prototype returned to us via the pre-processing stage of the build, chapter 4,, the next stage is to construct our object using the prototype. Here the builder does a few checks, firstly it looks to see if an object already exists under the name returned by the pre-processing stage. Names are unique, the name check simply involves the parsing of a "model" list looking for a match; whilst parsing the dependency tree is relatively fast and straight forward, a linear list of all objects is quicker to examine. The builder therefore maintains a lists of all objects currently within the dependency tree, since the builder is responsible for modifications of the dependency tree. If no object exists within this list under the name of the new object to be, the builder immediately proceeds with the construction.

Should an object already exist of the same name, the builder assumes a reassignment is taking place. For this to occur, the two objects must be compatible. Objects dependent upon the existing object are expecting a particular family of object, whatever the concrete representation may be, the dependent object is accessing the generic interface of a value, point, segment or other type. The new object must match this type otherwise the program will fail; the dependent objects contain a list of parent objects in the form of pointers to *Things*, see chapter 3, only they know what objects they really cast to. The new object must cast to the same type for the program to operate correctly. Also, the dependent objects only know how to convert their particular concrete representation into the generic representation. Altering a type of object they depended upon would

also cause them to fail. The builder therefore verifies the match of the existing and new objects families by comparing the results of the *family* function from the existing object and the prototype for the new object. Verification of matching families allows the builder to proceed with the construction as per the case of a new object.

5.1 Construction Via the Prototype

All prototypes have a *construct* function; being passed the variable name of the object the prototype is to create, along with an expression of the object's defining arguments, the prototype sets forth on its constructive path. The builder now acts as a resource to the prototype which, having the knowledge specific to its particular object, sets forth verifying any provided input in order for to ensure a successful object build.

Taking the input "p = vvpoint(x,y)", the *vvpoint* prototype will receive the name p and argument list of two tokens x and y. Firstly the prototype will check that x exists and is a value, to do this the prototype will access the builder that initiated the construction and enquire about the variable. The builder has a list of all variable objects that it has constructed, this list reflects every object in the dependency tree. Instead of a dependency tree search, the builder can search its own list for particular objects; this is quicker, the builder assumes responsibility for correlating this list with the objects of the dependency tree for every action taken upon the dependency tree.

5.1.1 Lookup of Object Variables

The builder has two generic functions, *resolveThing* looks through the builders list, a list of *Things* from which all objects are derived, searching for a named variable according to a given type of family; upon finding the named object, noting all objects are uniquely named, the builder can then check the family of the object is correct and return it to the prototype. The prototype, happy with this resolution, advances to the next argument it has to check in the example, y. Successful referral of these two variables allows the prototype to proceed with the

construction of the object, passing the referenced variables through as the arguments required to construct the object. Failure to resolve any of these arguments would cause the prototype to return a null pointer to the builder signalling the failure, the prototype would print useful messages to the user explaining the problem using generic functions within the template prototype which all prototypes inherit; constructing a prototype is thus made as simple as possible, requiring the implementation of the polymorphic *construct* function.

To facilitate use of *resolveThing* by the prototypes, a file is provided containing macros that supply the type of family required by the builder. Our *vvpoint* prototype will then use *resolveValue* to find if value x existed, the *resolveValue* macros supplying the value type to the *resolveThing* function of the builder.

5.1.2 Lookup and Automatic Manufacture of Object Variables

The second of the builder's functions is used more so by the prototypes; alternative to *resolveThing* is *resolveOrCreateThing*, mirrored with macros such as *resolveOrCreateValue*. If we are to ask the *vvpoint* prototype to construct "p = vvpoint(x,5)", the prototype will successfully look up x as before. However, the value *5* doesn't relate to a variable name. We could construct a variable called *five* automatically and make variable p dependent upon that, a better solution is to construct a variable name dependent upon the variable that requires it; house keeping is easier this way, should we delete p we then know to delete the automatically manufactured variable too. State of the automatically manufactured variable is also independent of the name, allowing us to change the value of this variable without confusion. The *resolveOrCreateThing* will automatically manufacture variables as required. To simplify prototypes, this function is used by default unless automatic manufacture is specifically undesired. Just as the call to the *resolveThing* macro supplied one of the construct arguments, being the name to lookup, the call to the *resolveOrCreateThing* macro supplies this same information. Here, however, it's the number *5* in our example, and is a nested declaration for construction. The *resolveOrCreateThing* macro also takes the name of the object being created by the prototype along with an identity tag; the macro supplies another tag indicating the family type and all this in-

formation becomes the name of the manufactured object. For our example, the manufactured name of *_macro*Tag*prototype*Tag*_name* becomes *_vy_p* indicating the manufactured object is a value for the *y* coordinate of object *p*. Function *resolveOrCreateThing* is told by the macro *resolveOrCreateValue* to look up value *5* which doesn't exist; the function then creates the manufactured name and asks the builder to evaluate "_vy_p = 5", this recursive action now entering the same construction cycle we initially ventured upon. The builder's pre-processor will deduce we want "_vy_p = int(5)" and the *int* prototype will construct an integer value of *5* which the prototype will return to the builder, for the builder to return to our initial prototype allowing construction. The recursive nature of the process allows use of shortcuts via the pre-processor at all nested stages. Manufactured names are guaranteed unique and interpretable to the user should they wish to modify their state.

5.2 What to do in the Event of Failure

Image we try to resolve "p = vvpoint(5,y)" which is very similar to the example before. The *vvpoint* prototype is found, it calls the *resolveOrCreateValue* macro which in turn calls the *resolveOrCreateThing* to create the manufactured object *_vx_p* as the automatically manufactured "x" coordinate value of *p*. The *resolveOrCreateThing* passes "_vx_p = 5" to the builder whose pre-processor determines the expression expands to "_vx_p = int (5)", the *int* prototype successfully creates the integer value object, returns it to the builder which returns it to *resolveOrCreateThing*. Here it is checked that a value was actually created, as it was, because *resolveOrCreateThing* was called by *resolveOrCreateValue* with the type set to *value*. All is successful and the automatically manufactured object is returned to the *vvpoint* prototype. Now this prototype calls *resolveOrCreateValue* asking for a value named *y*, the problem occurs if *y* doesn't exist. The *vvpoint* prototype is told that this variable doesn't exist, it thus exits after returning a null pointer to tell the builder that the object *p* failed in its construction. The resulting problem lies with the automatically manufactured value object that's sitting in the dependency tree. Whilst this may taint our house keeping, leaving unwanted objects lying about that could easily be cleaned up at a convenient moment, the failure exhibits itself if we were to consider that this point object, *p*, already existed. As it will be later seen, rather than try to modify the internal state of an object, it is

easier and more advantageous to create a new object and swap it for the old one. Therefore, if point p already resides in the dependency tree and we fail in our new declaration of p, we run the risk of the newly automatically manufactured variable erasing the old one; this would leave the dependency tree in an incorrect state because the newly manufactured variable would certainly be of a different numeric value.

5.2.1 Inherent Undo and Redo Mechanism

To resolve the possibility of leaving the dependency tree in a broken state, as a result of failed builds, we record the actions of the build after each operation; at the point after a prototype returns a successful construction, the builder has a newly created object. Should that object already exist, the builder removes the old object and substitutes the new object. Here we create a history object that states a replacement took effect, recording the removed and inserted objects. Should the object be a new addition, the history object states there was an addition and references the new object. Each recursive action of the automatic manufacture process results in a history object being tagged on to the end of a list of such objects; if a build fails at some point in the installation, the history list can be parsed in reverse order reinstating objects.

On completion of a successful build, this history list need not be discarded; by creating a list of history lists, it is possible to traverse this list and restore the model to any previous state. Travelling backward through the list of history lists, reinstating the previous state of the dependency tree by traversing the last history list backwards, we are performing an *undo* operation. Once restoration is completed, we can then remove the history list from this *undo* list and tag it to the end of a *redo* list. As a byproduct of the build fail mechanism, the user is now able to traverse through the various states of the dependency tree. This is an abstract mechanism, the undo features are inherited when the derived object inherits from the base *Thing* and as a result can be applied to any object type without type specific programming[12].

5.3 Additions and Replacements

Having determined, at the point the *GBuilderExpressionResolver* returned a prototype, whether an object was being added or an existing object was being reassigned, the builder will have recorded a respective *added* or *replaced* history object. If the object was an addition, the builder adds the object to the "model" list detailing all objects currently within the dependency tree. No modifications need be done to the dependency tree by the builder, the constructor of the added object handles this; for every parent object that it depends on, it adds itself to those object's child lists. This process is performed by the object itself because it knows what parameters passed to it should actually constitute dependencies. For each object it depends upon it calls *adoptParent*, a function within *Thing* thus inherited by all dependency objects; this function adds the calling object to the child list of each parent specified for *adoptParent*. For the example "p = vvpoint (x , y)", the point p will call *adoptParent* for parent x and parent y as follows:

```
// constructor for VVGPoint, passed the two x, y value parameters

VVGPoint::VVGPoint(GValue *x, GValue *y, NodeBuilder *nodeBuilder,
   const char *str)
  : GPoint(nodeBuilder, str)
{
  // adopt x and y as parents..
  adoptParent(x);
  adoptParent(y);
  _node = _nodeBuilder->newNode();
  // (re)construct the generic representation of the GPoint
  reco();
}
```

Replacement builds have a little added complexity, the old object needs pulling from the child lists of its parents and the parent lists of its children; the new object is then substituted into these lists, ensuring order of parent lists is maintained. However, the lists of the removed object are kept as intact as possible; this information allows the object to be reinstated within the dependency tree during an undo operation. The old object is removed from the model list, a history object

is created stating the replacement, removed object, and replacing object, then the new object is added to the model list. This process effectively deletes the old object from the dependency tree, without removing the dependents, and inserts the new object; normally dependents would be removed in a delete operation, they cannot exist because they reference the deleted parent. For house keeping purposes, any manufactured objects of the replaced object are deleted. The new object will have its own manufactured objects created before itself. Should any of these match the old object's manufactures by name, the recursive nature of the builder would mean that they would already have been manufactured and have replaced those objects; having been replaced, they would be out of the dependency tree and the builder's list of current objects. Deletion works only on object's within the current dependency tree, allowing deletion through the subject's parent list trying to delete any manufactured objects found. The subject cannot be deleted, it is already removed from the dependency tree, some of the manufactured parents are possibly removed from the dependency tree, had they already been replaced by the builder. Any remaining manufactured variables affected are removed if independent. This ensures that manufactured objects can't be deleted should the program user specifically have added objects dependent upon these. The program will never do this, however it's possible for the user to create an object with manufactured objects that the user then adds dependency upon. The following example illustrates the point:

```
# create a point that results in two manufactured variables..
p = vvpoint (1, 4)

# listing the current model..
list
_vx_p = int(1)
_vy_p = int(4)
p = vvpoint (_vx_p, _vy_p)

# add a dependency upon a manufactured variable..
p2 = vvpoint (5, _vy_p)

# remove p, _vy_p cannot be removed as p2 depends upon it..
rm p
list
```

```
_vy_p = int(4)
_vx_p2 = int(5)
p2 = vvpoint (_vx_p2, _vy_p)

# remove p2 and all will be deleted..
rm p2
```

House keeping ensures tidiness by removing any manufactured, and independent, parent object regardless of whether it was manufactured by the object being deleted. Therefore, deleting the main subject with *deleteUp* results in *deleteUp* being called upon all parents. If they are manufactured and still within the model, *deleteUp* will remove them and act upon their parents moving up the dependency tree actively removing all redundant objects. Manufactured objects only are removed in this process, other objects, whilst potentially being independent and unused, were specifically put there by the user.

With the replacing object now inserted into the dependency tree, another generic function provided by the *Thing*, is used to flag all the dependent objects as *pending*. This means, as we've replaced an object they're dependent upon, that their state now needs revision. We don't have to update the states' of these objects until it's actually required, until then the pending flag ensures that their states will be revised when time necessitates.

Finally, the builder calls a polymorphic function within the builder itself called *thing_evaluated*. The generic builder doesn't implement this function, its purpose is to allow inheriting builders access to the individual events that occur in the build process. An inheriting builder would be able to process events, examining the built objects in order to perform object specific post-processing upon them. This allows the base builder to remain as generic and reuseable as possible. Taking the *thing_evaluated* function, which supplies a pointer to the object just built, an implementing function in the inheriting builder could construct a list of all evaluated objects. Easy identification of modified objects is now possible without scanning of the dependency tree. Additional "hooks" into the build and model update are detailed in the following chapter.

5.3.1 Advantages of Replacements

Initially a *reconstruct* mechanism was tested with objects; when an expression was entered matching an existing object variable, the parameters of the expression were passed to the reconstruct function of that object. This worked within a limited scope, for instance a real value successfully changes state; a line's length, dependent on that value, will thus change through the dependency tree. The line is dependent on a value for its length, it can only be modified by changing the value it depends upon. There is little the reconstruct mec within the line. The more complex an object, the less scope for modification. We could not make that line depend on completely different object types, we could not turn the line into an arc even though the line and arc, as segments, are indistinguishable to dependent objects.

Actually removing an object and replacing it allowed insertion of the new object with modified state, equivalent to reconstruction for the simplest of objects like real values. Dependent objects saw the same type of object, with a different state, generic interfaces allowed objects to be interchanged provided they had the same interface. There was however no necessity to maintain the same parents during this interchange, since the newly inserted object would hook itself into the child lists of its parents; the old object simply needed to be unhooked from its parents. This made it possible to replace one object with a like object dependent on completely different objects, the children still seeing the same generic type of object. Different concrete representations of a family could be interchanged, the circle dependent upon centre and radius being switched for a circle dependent on three points, or the circle being switched for an arc and any other segment. The user of the program is able to modify the model in a much wider variety of ways, changing the order of dependency as well as the states of variables.

Had the *reconstruct* method been used, manufactured variables would have been reconstructed during the reassignment process; had the process failed part way, there would be no way of restoring the model's state without objects themselves knowing how to undo their reconstruction. Replacement of objects ties in greatly with the history providing *undo* and *redo* mechanisms. Object state and Dependency tree manipulation are two advantages, an inherent undo mechanism is another. Objects need no programming for remembering states, this creates simpler objects and allows the history mechanism to apply to any objects. There

is no limit to the depth of model change attainable other than memory size.

Chapter 6

Build Post-processing, Updating the Model

The following chapter concentrates on the builder's *updateModel* function. This function signifies the point in the build where the principal object has been completed, either successfully or unsuccessfully, and the builder is now realising the effects this has on the entire model. Post-processing of the model is accommodated through a few polymorphic functions. The generic builder is designed to be just so, there is nothing specific to object types; any processing specific to objects is designed to be implemented in builders inheriting from the generic builder. These builders can implement polymorphic functions allowing access to key events in the build and model update process. One such "hook" into the process has been introduced in the previous chapter, *thing_evaluated* is a function called after every object is built. The *PMGBuilder* used by the program implements this function in order to maintain a list of all objects inserted into the model for that operation.

6.1 Updating The Model

Function *updateModel* is self-contained, designed to be called whenever actions are taken upon the model. Any model change involves manipulation of the dependency tree along with a corresponding reflection within the list of objects current

to the dependency tree, known as the model list. Any change is immediate, the dependency tree and model list are altered by the modifying routine, it is the history that primarily concerns the model update. This history lists all additions, replacements and deletions taken upon the model. By default the builder is interactive, updating the model after every assignment entered by the user such that each individual step the user takes is stored separately in the undo mechanism. If the user added an object, then deleted an object, undo would restore the deleted object the first executed time, then remove the added object when executed a second time. Reading input from a file usually involves the creation of many objects, here the builder is placed into a non-interactive mode and update of the model is done only when all objects are read from file. Now the history for the file opening contains as many addition entries as there were objects in the file, the update stores this as one action and undoing this will remove all these objects simultaneously. Likewise, the rename command replaces objects with renamed counterparts in one action; as renaming involves manufactured children, explained next, and parents, the names a composite of the name being changed, several objects exist in the history of this change. Therefore an update affects as many objects that were added, replaced, and deleted, since the last update.

In the builder's non-interactive mode, *evaluateExpression* doesn't call *updateModel*. It is the job of the program, such as a file open command, to call *updateModel* at the appropriate time, usually before the program returns to an interactive state. In calling *updateModel* a true or false *success* argument is supplied, telling *updateModel* whether the action succeeded or failed respectively. In an interactive mode, *evaluateExpression* calls *updateModel* passing the built object as this argument; if successful the pointer to the object counts as true, if unsuccessful the null pointer passed counts as a false. On success *updateModel* will continue post-processing of the model, otherwise it uses the history to reverse any changes made upon the model before discarding the history.

6.1.1 Additional Object Builds

We have discussed automatic manufacture of parent objects in the previous chapter, performed prior to the build requiring these parents in order to manufacture objects in the order of dependency. Now we discuss cases where automatic manufacture of children is required because this is the first action undertaken by

updateModel.

Taking the segment object, a line segment can be drawn between two points as in figure 6.1:

Figure 6.1: Line Segment Constructed From Two Points

The dependency of the line segment, figure 6.2, shows by definition that two points define its extents. We now draw the different line segment and dependencies of figures 6.3 and 6.4 using a starting point, angle, starting and ending length projection:

This example line segment is not bound by any points, nor are there any points with which to anchor subsequent objects. This segment is quite useless unless intersected and used purely with intersection points, the definition of a segment thus stipulates that points should terminate all segments in order to provide the useful anchor points. The best way to implement this is for these particular segment types to construct the necessary points, rather than have a builder that needs knowledge of what objects needs special requirements; we therefore maintain as generic a builder as possible.

Initially these objects constructed the necessary anchor points within their own constructors. This caused a problem because the dependent points were recorded by the builder as being constructed before the segment they actually depended upon, this happening because the builder records objects as a prototype returns the successful build. Repercussions of this were in reconstruction of the dependency tree during undo and redo operations, whilst achievable it was ultimately found to be much simpler if objects entered and left the tree in order of their dependency. Moving the construction of the dependent anchor points into the prototype would also fail for the same reason, object creation had to be postponed until the prototype had exited.

A solution to the problem was found by taking the expressions, used by the segment to define dependent anchor points, and supplying them to the builder as

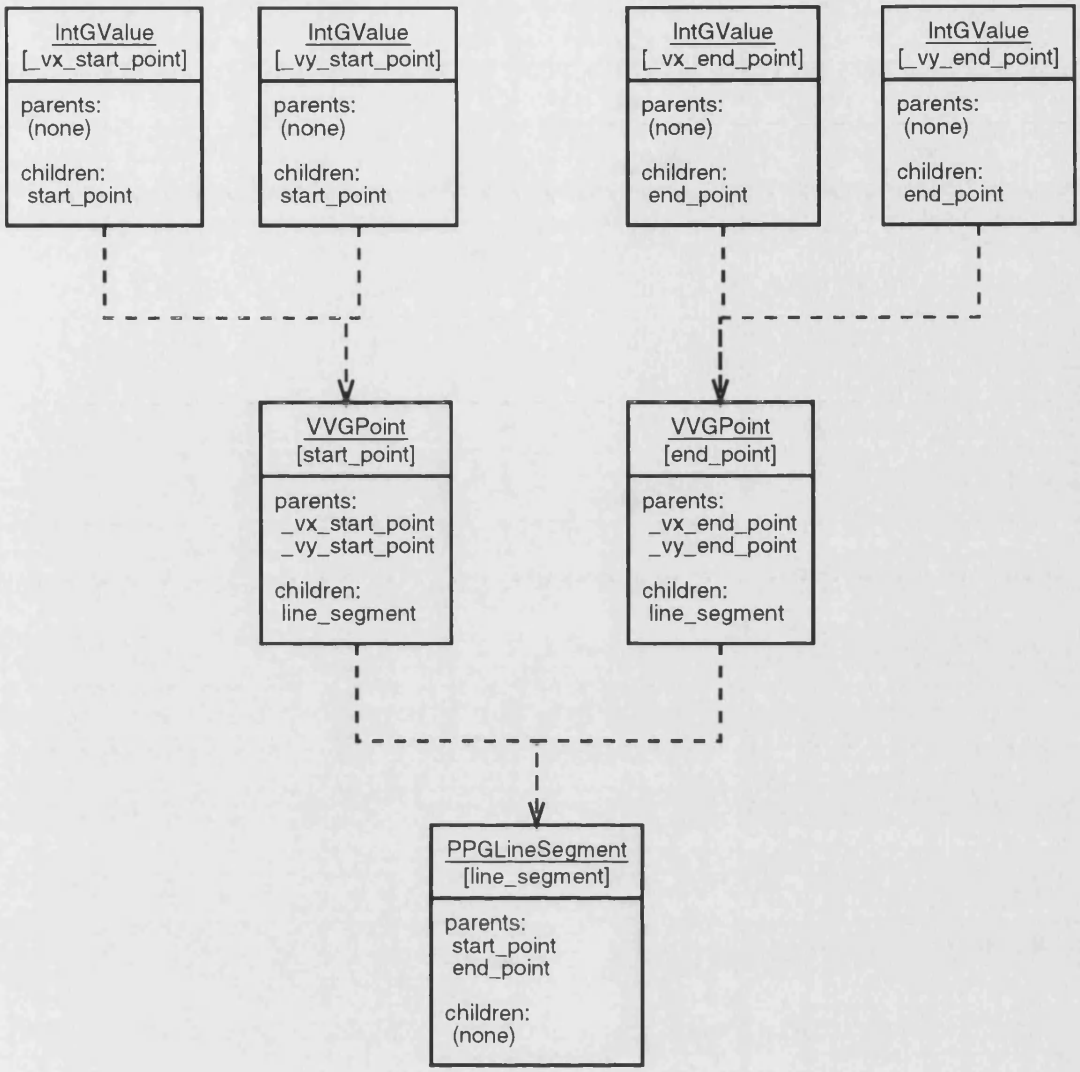
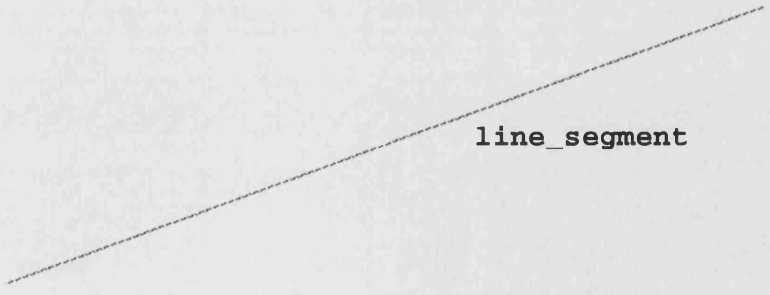


Figure 6.2: Line Segment Dependency on Two Points



projection_point

Figure 6.3: Line Segment Constructed From Centre Point, Angle and Length Projections

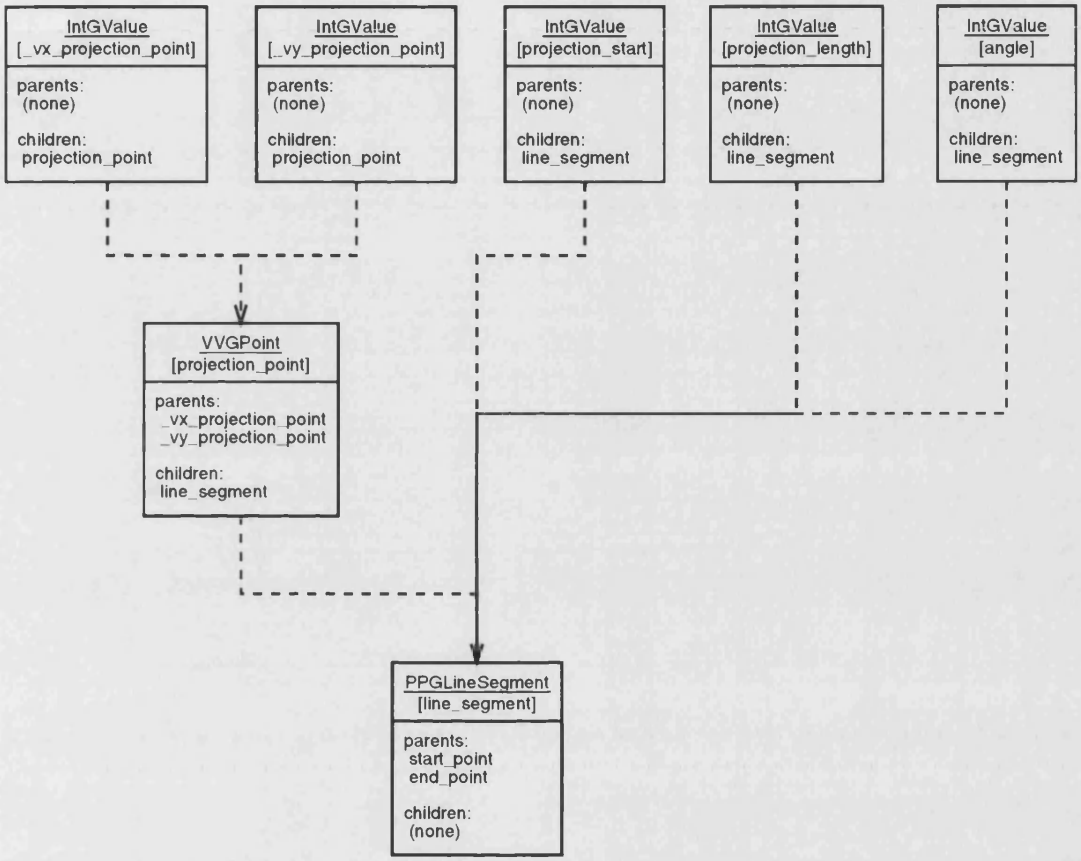


Figure 6.4: Line Segment Dependency on Centre Point, Angle and Length Projections

appended builds; the builder maintains this list of expressions and on successful completion of the segment build, the builder would process these expressions and all objects would be processed in the correct order. The mechanism would work if appended builds yet further supplied appended builds too.

For a segment to manufacture termination points, it would supply the following expressions to the builder's *appendExpressionForEvaluation* function. The two points take manufactured names, produced via the builder's *createLabel* function, to denote their internally manufactured nature:

```
# an spoint takes a segment,
# binding to the position of its specified point.
# start point of segment,
# the '0' becomes an automatically manufactured value..
_ps_segmentName = spoint ( segmentName , 0 )
# end point of segment..
_pe_segmentName = spoint ( segmentName , 1 )
```

The following figures show the final construction, the segment with its termination points, figure 6.5, and the corresponding dependency tree, figure 6.6.

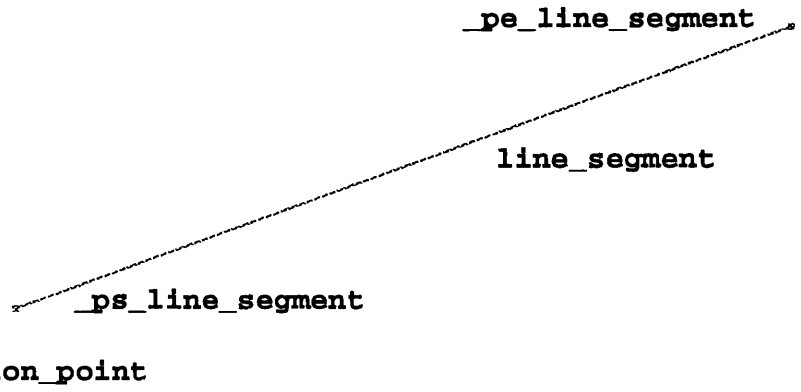


Figure 6.5: Line Segment With Constructed Anchor Points

If any appended build should fail, the build process is now considered unsuccessful; no further appended expressions will be processed and *updateModel* will proceed as it would if told previous actions had been unsuccessful, using the history to restore the model to its state subsequent to the last update. Having processed these appended expressions in a non-interactive mode using the

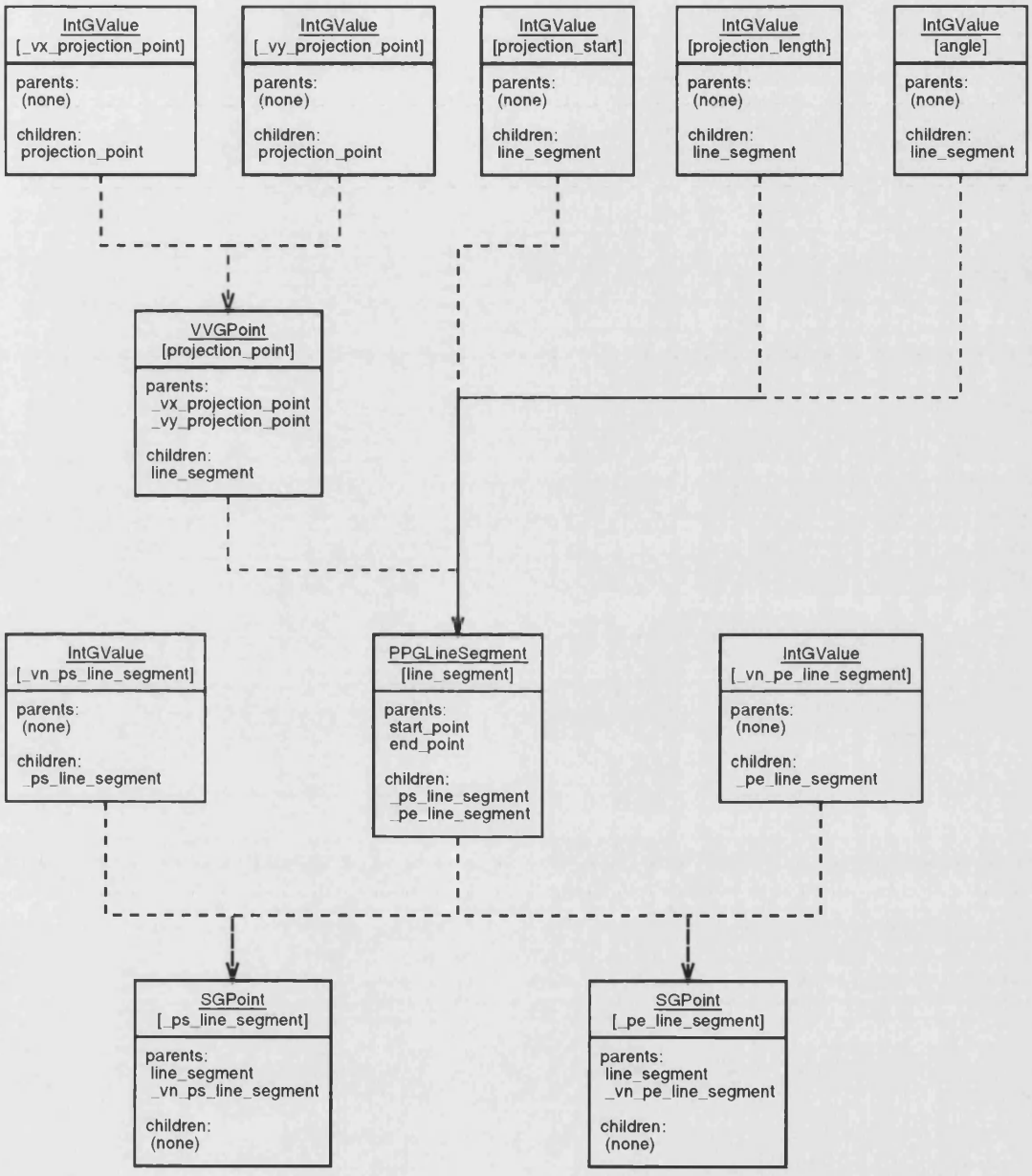


Figure 6.6: Line Segment Dependency of Constructed Anchor Points

usual expression processing of the previous chapter, the builder will have these additions tagged to the end of the current history list.

6.1.2 Beginning the Update for Post-processors

The generic *updateModel* function has now added any objects required by previous builds, removal of related objects has been performed. Now is the perfect time for post processing of the model, any changes resulting from this will still be added to, and subsequently processed from, the history.

The Generic Builder

The generic *GBuilder* has been kept free of processing specific to particular objects. Pre-processing within the builder used knowledge of low-level objects only, these being necessary for any build application, with the prototype mechanism being used to match higher level object types, keeping the knowledge in the prototype rather than the builder. Even then, this processing was separated into the *GBuilderExpressionResolver* object; first to allow pre-processing without any object knowledge with the use of a slimmed down expression resolver, and secondly to allow expansion upon the object knowledge, and thus the short cuts in expressing them, without the need to touch the builder.

Extending the Generic Builder's Capabilities

The following functions are polymorphic in the *GBuilder*, inheriting from the *GBuilder*, implementing any of these functions, allows processing on objects in a sane manner protecting against the unusual effects that can occur when modifying the dependency tree.

thing_evaluated(Thing *thing_evaluated) :

called after an expression is evaluated and an object has either been added or replaced. Used by the *PMBuilder* to maintain a list of modified objects.

model_update_begin(bool evaluation_was_successful) :

actually called just after model update begins, once the model has the post processing performed that might add or remove objects. The model has reached a steady state and the resident objects can be examined safely. The *PMGBuilder* uses the list of evaluated things, obtained from the previous function, to identify segments and components whose position has changed; this allows these objects to be re-examined for intersections with like objects. This is only done if the operation was successful, *evaluation_was_successful* set to *true*.

model_update_adding(Thing *thing_evaluated) :

added and replaced objects actually exist within the model before the update procedure, the history objects describing these actions are simply analysed to cement this fact; their function mainly being in restoring the model to its original state upon failure. However, the *PMGBuilder* uses this function to add any such displayable objects to the display. The action updates a flag that ultimately determines that the display should then be redrawn.

model_update_removing(Thing *thing_evaluated) :

removed objects, or those replaced, identified within history objects, cause this function to be called when the update cements this change. The *PMGBuilder* then removes any such object from the display, should it be of the displayable kind. The action flags the display for later update.

model_update_end(bool evaluation_was_successful) :

finally, a function called when absolutely every update has been performed and the model has reached its final state. The *PMGBuilder* uses this event to display information about the current state of nodes and elements, plus it tells the display to redraw itself if *model_update_adding* or *model_update_removing* changed the number of displayed objects.

The *updateModel* function now calls *model_update_begin* to allow further analysis of the model. Any resulting changes will be included in the history of this operation, grouped together as one long list of history objects marking all changes since the last model update. The nature of the post-processing is likely, as in the *PMGBuilder's* case, to validate many objects within the model. Intersection of segments, performed by the *PMGBuilder* causes many intersection points to be added or invalidated. Until this stage is finished, the model is far from complete.

6.1.3 Dealing With Invalid Objects

If we take two intersecting segments, defined as follows and illustrated in figure 6.7.

```
# draw one line segment from point 0,0 to point 1,1..
l1 = ppline(vvpoint(0,0), vvpoint(1,1))

# intersect with another line segment from point 0,1 to point 1,0..
l2 = ppline(vvpoint(0,1), vvpoint(1,0))
```

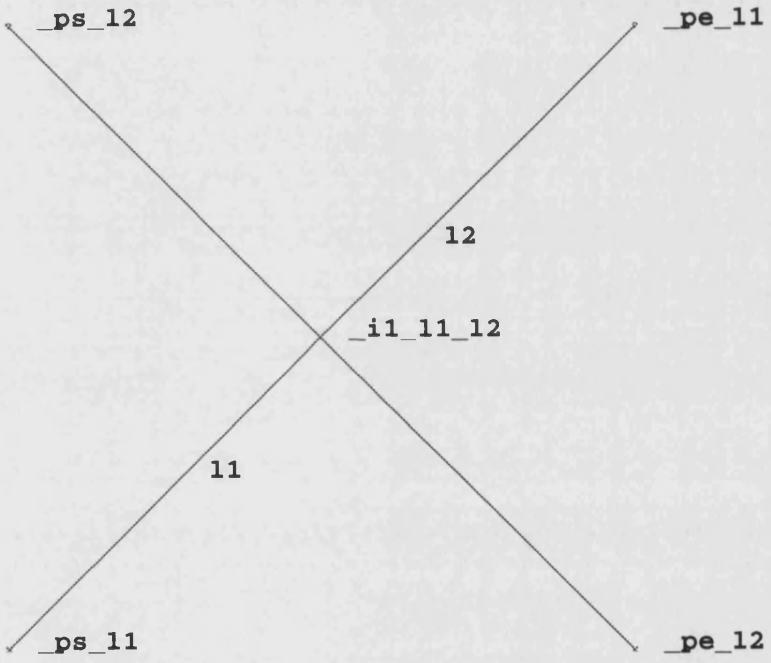


Figure 6.7: Two Intersecting Segments

Now we take line *l2* and shift its position so it no longer intersects with line *l1*:

```
l2 = ppline(vvpoint(0,1), vvpoint(0.4,0.6))
```

Figure 6.8 reflects this change, the intersection point *_i1_l1_l2* between these two line segments is rightly nowhere to be seen; these two lines no longer intersect

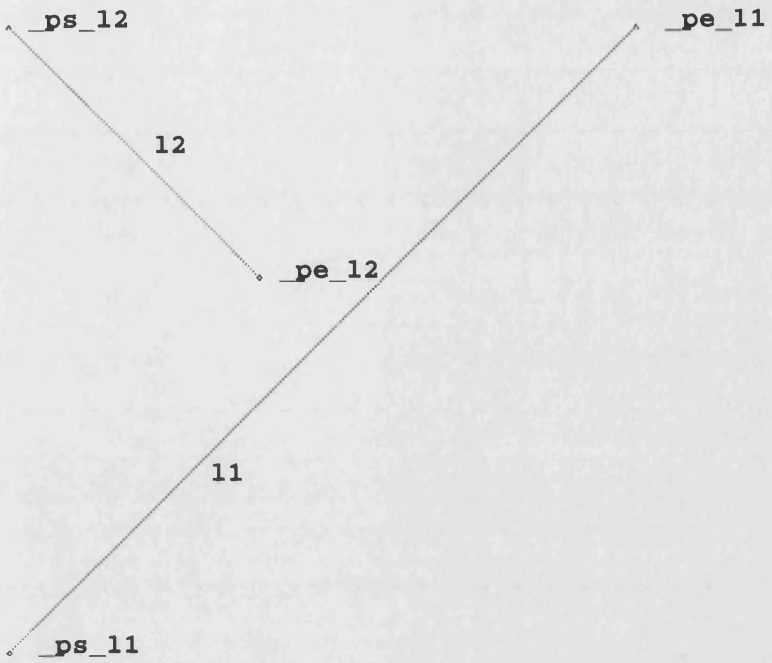


Figure 6.8: Two No Longer Intersecting Segments

and this point has no idea of what position it should take, detecting this it has returned a *false*, failure, value from its reconstruction, *_reco*, function. The generic *reco* function within the *Thing* has thus marked the intersection point invalid, it would also have marked any dependents invalid too.

The builder's final parse of the model, before processing the history objects, checks for invalid objects. Invalid objects can quite happily reside within the dependency tree. An object's state is checked, either through the invalid flag or more appropriately through the *reco* function to ensure the state is updated if pending, before it's used; invalid objects are never used, they are therefore never displayed, selected, or acted upon by any other operations except a model list or save. The user will therefore know nothing of invalid objects unless noticeably missed. The builder will thus check for invalid objects and alert the user; this allows them to ignore this state, later they can alter the model to re-validate objects, otherwise they can be deleted if the invalidated branch has become so because it is redundant. Currently only the deletion process is implemented, future work, chapter 9, would include an interactive element which ideally incorporates a graphically interactive process.

6.1.4 Cementing Changes

With invalid objects potentially deleted, finally all additions and deletions have been performed upon the model. Had there been a failure along the line, or in the main build itself, the history will be used to revert the model to its previous state, subsequent to the last update. Otherwise, all changes stored in the history objects will be parsed and filed away to allow later undoing of the changes; having said that, it's possible to pass a flag to the builder, upon its creation or a later, disabling undo operations. Here the undo mechanism still works to the point that failed build operations can be reverted, history of the changes of a successful build are simply discarded so that undo and redo operations by the program user are not supported.

The History

Every change has been stored in a history object, this *ThingUndo* object simply consists of three fields incorporating an action descriptor and two *Thing* pointers. The action descriptor describes the operation, add, delete, or replace, then two pointers point to "was" and "is" objects. Additions set the *is* pointer to the added object, deletions set the *was* pointer to the removed object, replacements set the *was* pointer to the replaced object, the *is* pointer being set to the replacing object. From this pattern it can be seen that the action descriptor is redundant, the pointers amply describing the action, however the descriptor is maintained should additional actions be added in the future.

Within the builder is a *ThingUndoList* called *_history*; as soon as modifications begin on the model, the describing *ThingUndo* objects are added to the *ThingUndoList _history*. This list is now going to be parsed, if user undo is enabled then a new *ThingUndoList* is created for the addition of parsed *ThingUndo* objects. Once every *ThingUndo* object has been removed from the *_history*, parsed, and then added to the new *ThingUndoList*, the new list can be added to the builder's *_undo* list; this is a *ThingUndoListOfLists*, lots of lists do seem to be involved in order to keep track of events! A redo list accompanies the undo list; when an undo takes place upon the model, all the information within the last *_undo* list's last *ThingUndoList* list is parsed in reverse order. Once the model is restored to

it's previous state, this *ThingUndoList* is removed from the end of the *_undo* list and added to the end of the *_redo* list. Should the model be changed after this, the *_redo* list must be erased; the *_redo* list references changes in the future that may no longer be possible to make if the current change affects the objects they reference.

Successful Build

The following describes the history parsing in view of a successful build:

- If user undo is enabled, create a new *ThingUndoList* for adding *ThingUndo* objects to as processed.
- Pull the first available *ThingUndo* object from the *_history* list.
- For replacements and removals, call *model_update_removing* for the *was* object pointed to by *ThingUndo*.
- Likewise, for replacements and additions, call *model_update_adding* for the *is* object pointed to by *ThingUndo*.
- If user undo is enabled, append this *ThingUndo* to the new *ThingUndoList*. Otherwise, delete any removed or replaced object by deleting the *was* object and delete the now redundant *ThingUndo* object.
- Once all the history is parsed, add the new *ThingUndoList*, if user undo is enabled, to the builder's central *_undo* list. As a result of this, the redo history must be cleared as it acts upon a model whose state is now different.

Unsuccessful Build

- Pull the first available *ThingUndo* object from the *_history* list.
- Use the *ThingUndo* object to reverse the action accordingly, without deleting any object pulled from the model.
- Add any returned object to a list of deletions, a *ThingList*, for later deletion. The now redundant *ThingUndo* object is deleted.

- Once all the history is parsed, delete all the objects in the deletions list. We couldn't delete these objects at the point the *ThingUndo* object was processed because the undo process doesn't just affect this object; the dependency lists of this object's parents and children would be affected, these objects may have also been processed and accordingly deleted. Trying to modify the dependency lists of deleted objects would be rather dangerous. Now all processing is done, all can be deleted safely.

The final stage of the model update calls the *model_update_end* function allowing inheriting builders the opportunity to hook extra processing into this final stage. The *PMGBuilder* used by the program uses this event to output some useful debugging information regarding elements and nodes, the display is also updated should objects have been added to, or removed from, the display.

6.2 The Specialised Parametric Model Builder

Here follows an explanation of the *PMGBuilder* that inherits from the generic *GBuilder* in order to provide specialised processing upon the model of the electrical machine. Having already introduced the functions the generic builder provides in order to implement specialised post-processing on a model update, the following serves to explain the nature of the *PMGBuilder* type builder and its post-processing in the context of electrical machine design.

6.2.1 Additions To The Electrical Machine Model

Any object that has been evaluated, this covers additions and replacements to the model of the electrical machine, is added to a list for later use. The specialised builder used for electrical machines knows of new evaluations by implementing the polymorphic function, *thing_evaluated*, which the generic builder calls after each evaluation; this function provides a pointer to the evaluated object.

6.2.2 Starting Electrical Machine Specific Post-processing

The generic builder performs some post-processing itself upon the model; at the point this is finished, we now have in place all the desired objects, and repercussions of these, caused by the evaluation the builder was initially asked to complete. Now a *model_update_begin* function is called to allow post-processing of the static model specific to a specialised builder. For electrical machines, this builder determines intersections between segments and also components.

Intersecting Segments

All modified objects, identified through the list formed with the *thing_evaluated* object, are flagged as modified along with objects dependent upon these; this is through the use of *flag* and *flagDependents* functions of the *Thing* interface of chapter 3. The builder has a linear list of all objects held within the dependency tree, this allows easy identification of objects without the need to parse the dependency tree avoiding multiple references to the same object. This list is now parsed and any segment identified as being modified is added to a list of pending segments, the same is done for components. This methods avoids the identification of objects no longer within the dependency tree which are in the undo history; such objects may exist within the list obtained through the *thing_evaluated* if actions replaced them and then deleted them, entirely possible considering multiple evaluations may be offered to the builder for inclusion in one model update; in other words, perform all these evaluations sequentially and store as a single change so that one undo of the model will revert all changes.

Now segment intersections are examined. The first segment is pulled out of the pending segment list just constructed. That segment's *reco* function is called to ensure it's state is up to date; segments have been identified through being modified or dependent upon a modified object, marking them as pending reconstruction. Changes do not take immediate effect upon dependent objects within the model, only when they are used is their state updated by calling the *reco* function. This allows for more efficient operation, only incurring the overhead of updates when objects need to be used. As a bonus the *reco* function also returns the validity of the object, we check this to ensure we have an object worth testing.

The first segment is now intersected against all other segments held within the builder's linear list of objects contained within the dependency tree, the "model". Those segments are first checked for validity by calling their *reco* function, noting that this simply returns the validity if they're up to date, not too expensive on processing, an additional check is then done to ensure this intersection didn't take place before; in order to provide consistency in the labelling of intersection points we produce the name "_i{*intersectionNumber*}_*segmentName*_*otherSegmentName*" where *segmentName* alphabetically precedes *otherSegmentName*, this way we result in the same point name irrespective of whether we intersect *segmentName* with *otherSegmentName* or vice-versa. For a circle and line segment, "circ1" and "line1" say, two intersections could result giving the names _i1_circ1_line1 and _i2_circ1_line1. All processed intersections within this update are stored using the name of the intersection point, we can now check for this name in case a previous intersection involved the intersection of these two intersections in reverse.

With checks completed we now intersect these two segments by asking one to *intersectWith* the other. Chapter 7 details how two segments are intersected when these segments are referenced through an abstract *GSegment* interface which hides the true circle, line, or arc implementation. If one or more intersections were found, an intersection point is constructed for each by creating an expression for the builder to process through its *processExpression* interface. The names of these intersection points are then added to the list of processed intersections in order to avoid duplicate intersections later.

6.2.3 Adding and Removing Building Block Objects

With all post-processing performed, the generic builder sets about cementing changes in its undo history. This process is completed regardless off the success or failure in the desired evaluation. The mechanism either restores the model to its initial state if a failure occurred, otherwise the effect of every addition, replacement, or deletion is recorded for prosperity. A failed evaluation at hand, nothing is to be done by the specialised builder as the initial model state is restored. On success, every object that has been added to the model is announced using the *model_update_adding* function; every removed object is announced using the *model_update_removing* function. An object replacement is effectively an object removal and then an addition, the replaced object is removed and the

replacing object is added with *model_update_removing* and *model_update_adding* being called respectively. The specialised builder uses these events, taking the announced objects and examining them for use of the *GeometricThing* specialisation to the *Thing* object. This effects an inheritance, from *Thing*, adding functions to display and detect objects with a geometric presence in a graphical environment. Any such object will be accordingly added or removed from the display by calling the respective add or remove functions a display has. *GeometricThings* can be seen in chapter 7.

6.2.4 Finalising Changes To The Electrical Machine

When *model_update_end* is called the model's state is final. We use this event to update the contents of the display, the change of any displayable object having set a signifying flag in the *model_update_adding* and *model_update_removing* functions. As a check, the management of node and element references is verified. This system is designed to maintain the ownership of a node or element with a particular object, dependent objects then referencing inherited nodes and elements. This system aims to alleviate replication of nodes in the final node and element output used to solve the machine's desired properties. When designing higher level building block objects, the management of nodes has been subject to a few "programmer errors" in the past. As reference builders maintain lists of all the allocated nodes and elements, see the end of chapter 7, we can quickly parse the list and look for out of place references. This point is the best place to do this since this builder knows of these reference builders and the model changes have reached a state of completion.

Chapter 7

Building Blocks of an Electrical Machine

Chapter 2 introduced the building blocks used to construct an electrical machine. We now discuss these objects in greater detail.

7.1 Communication and Representation

Lower level objects, such as values, are likely to be reused more than higher level objects, being dependent on more objects thus specialising them more. For this reason, lower level objects are split into two parts. The first part is communicative, being derived from the *Thing* of chapter 3, providing the generic interface for that object family, this is the interface seen by other dependent objects expecting a particular family of object, the *GValue* in the case of values. This part also provides the base for object inheritance within this family of object. Objects derived from this provide different dependencies on other objects, these *concrete representations* allow the generic object, the *GValue* for instance, to be defined in a number of different ways. Each concrete representation provides a different definition of the generic object, translating the data of other objects, which they depend upon, into the generic representation for view by the world through the generic interface. For higher level objects, the generic interface and representation are provided by the same object; the generic interface is usually just a way of

providing access, in a safe and protected manner, to the generic representation. Concrete representations, the objects that actually exist within the model, have already translated their parent objects' data into the generic representation, the interface they've inherited provides access to this. Thus for higher level objects, where the complex nature of the generic representation makes it less reuseable, the representation is also maintained by the communicative object providing the interface.

Lower level objects maintain their generic representation as a separate object within the communicative object. This collaboration allows the representation to be passed to other objects as a copy, the data represented may be manipulated through functions provided by the generic representation. In the case of values, the generic representation is provided by an *XValue*; this object encapsulates the representation, ensuring other objects access it through the "correct channels" by its interface. This allows the representation within the object to be changed, perhaps to optimise behaviour in some way, maintaining external compatibility by ensuring the outside world still sees the same interaction via the interface. The functions provided to manipulate the representation ensure that this is done so correctly, also minimising external functionality, and its repetition, by encapsulating it within the object. For values, as an example, this method has proven successful with the transformation of their representation from a single floating point number into real and imaginary floating point parts. The transformation maintained compatibility by translating this information in a way that kept the interface consistent with the previous version, access to the extended functionality was then provided through additional interface functions.

7.2 Values

The "roots" of the dependency tree, describing the model of an electrical machine, will always take the form of floating point or integer values. These two types are effectively the same, using the same representation, the integer type is simply used when indexing is performed. Their independence, they depend on no other objects and are without parents, places them at the root of the dependency tree; therefore they always provide the basis of the parameterisation and can identify all other objects within the tree, these being their children. Such values are

Public Methods	
virtual	**~GValue** ( )
double	**value** ( ) const
double	**mag** ( ) const
double	**ang** ( ) const
double	**real** ( ) const
double	**imag** ( ) const
XValue	**xvalue** ( ) const
bool	**canWrite** ( ) const
virtual void	**outX** (ostream &out)
const char*	**family** ( ) const

Table 7.1: GValue Public Interface

thus used when saving the model in order to ensure the arguments, other objects defining dependent objects, are saved before themselves; that way, when reading in a model, the defining objects are established within the model before they are referenced.

Table 7.1 shows the *GValue* interface. As mentioned earlier the representation for a *GValue* is held within the *XValue* object, table 7.3 illustrates this as being the sole item of data possessed by this object. The *GValue* is therefore an entirely communicative object, inheriting this communication from the *Thing* of chapter 3; as a result a few of the functions illustrated in table 7.1 are implementations of polymorphic functions inherited from the *Thing*. Function *canWrite* tells the outside world whether this object is capable of printing some information about its state, *outX* actually performs this function so *canWrite* would return *true* and *outX* would simply print the numeric values from the representation of real and imaginary components. The last function, *family*, is appropriately implemented by the *GValue* as all concrete representations inherit from this, it communicates the string of "value" as its family type; this becomes useful when replacing an object within the dependency tree with another of, what must be, the same family, explained in chapter 5. These three functions will be found to be implemented amongst all the object types detailed within this chapter.

A protected function, as shown in table 7.2, allows access only to derived, or inheriting, classes, as opposed to the world wide access public functions provide. Within table 7.2 is the constructor of the *GValue* object. This terminology is

Protected Methods
GValue (const char *str)

Table 7.2: GValue Protected Interface

Protected Attributes
XValue _xval

Table 7.3: GValue Representation

specific to the C++ programming language, explained best by the creator of C++ himself[18][19]. This function builds the object when a new instance of a value is required, the protected nature shows that values can only be built using one of the derived, concrete, types; derived types have public constructors, they can be built by anything and pass, in this case, the name of the object to the *GValue* object, to which they have access. Now moving back to the public methods of table 7.1, those remaining, barring the destructor *~GValue* which is responsible for closing the object down correctly, provide various methods of access to the generic representation. Most geometric use of values, as coordinates for example, use the *value* function to gain access to the real part of the complex value representation. Other uses, such as the setting of complex voltages, can access either the cartesian or polar representations of the complex value through the other functions.

7.2.1 Value Representation

Finally, the all encapsulating *XValue* representation itself can be extracted through the *xvalue* function to allow the passing of the representation as a complete object; the public interface of this representation can be seen in table 7.4, many of the functions are identical to those within the *GValue* interface which simply call these counterparts. Other functions are responsible for initialising the object and changing the values held within the representation.

Public Methods	
	XValue ( )
	XValue (double val)
	XValue (double real, double imag)
	XValue (const XValue &val)
bool	**complex** ( ) const
double	**value** ( ) const
void	**setValue** (double real)
void	**setValue** (double real, double imag)
void	**setValue** (XValue val)
double	**real** ( ) const
void	**setReal** (double real)
double	**imag** ( ) const
void	**setImag** (double imag)
double	**mag** ( ) const
double	**ang** ( ) const
void	**neg** (XValue val)
void	**add** (XValue a, XValue b)
void	**sub** (XValue a, XValue b)
void	**mult** (XValue a, XValue b)
void	**div** (XValue a, XValue b)
void	**power** (XValue a, XValue b)
void	**cross** (const XPoint &p1, const XPoint &p2)
void	**dot** (const XPoint &p1, const XPoint &p2)

Table 7.4: XValue Public Interface

Protected Attributes	
bool	_complex
double	_real
double	_imag
double	_hyp

Table 7.5: XValue Protected Representation

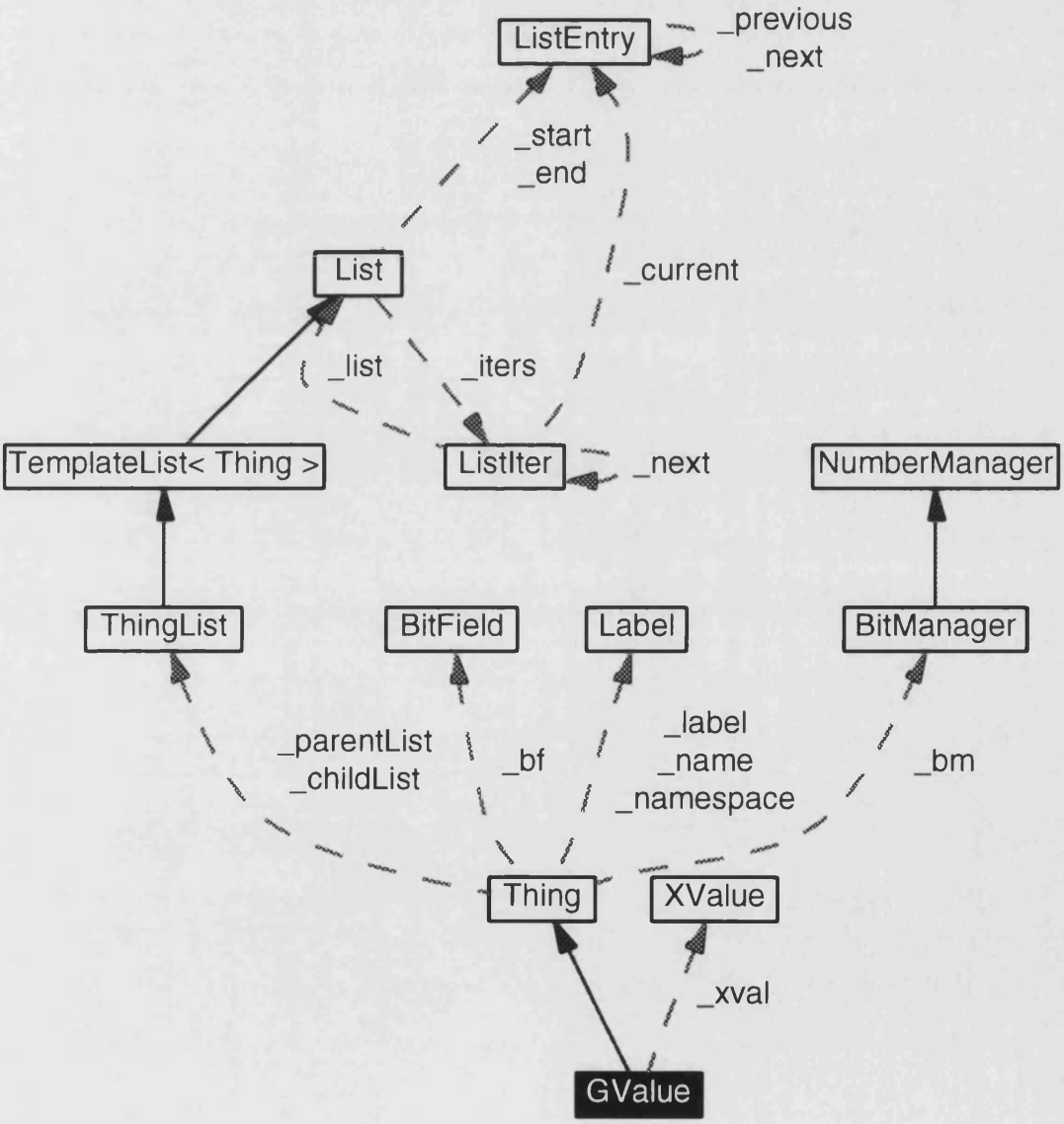


Figure 7.1: Collaboration Diagram For GValue

7.2.2 Inheritance or Aggregation?

With the use of Doxygen[20], a program that directly examines source code in order to determine an object's dependencies and collaborations, we can very easily construct pictures showing the relationships involved between the value objects. Figure 7.1 shows the inheritance of *GValue* from *Thing* using the solid line, the dotted line shows that a *GValue* contains an *XValue*. In object oriented terms there is a big conceptual difference between an object inheriting another object, and thus being a specialisation of such an object, and object aggregation, having such an object[21]. For values, either inheritance or aggregation would work and the *GValue* could arguably be an *XValue* or contain one. The deciding factor comes from maintaining some uniformity across our building block objects; the segment family has members of several groups, namely circles, lines and arcs, mirrored in the representation structure used. A *GSegment* must contain an *XSegment* because the abstract *XSegment* may be one concrete representation of many derivatives of *XLineSegments*, *XArcSegments*, and *XCircleSegments*. The *XSegment* representation a *GSegment* contains may also mutate into different segment types, for example when a three point arc's points form a straight line and the arc becomes a line; this is done by allowing this segment type two switch between an arc and line representation, the segment contains two segment representations and is something of which multiple inheritance is incapable of fulfilling.

7.2.3 Concrete Representations

Again we can use doxygen to extract and illustrate the inheritances, now including the concrete representations that form the manufacturable objects for use by the designer. Figure 7.2 allows us to show every value that currently exists, stressing again that additional objects may be easily added without complication, thanks to the object oriented design, without fear of breaking any of the existing functionality and without the need to modify any existing objects. Table 7.6 shows the *AddGValue* object's public interface; it consists of a constructing function, by which this object is called in order to create an instance of it, and the *type* function which uniquely identifies the object by the name of "add". The

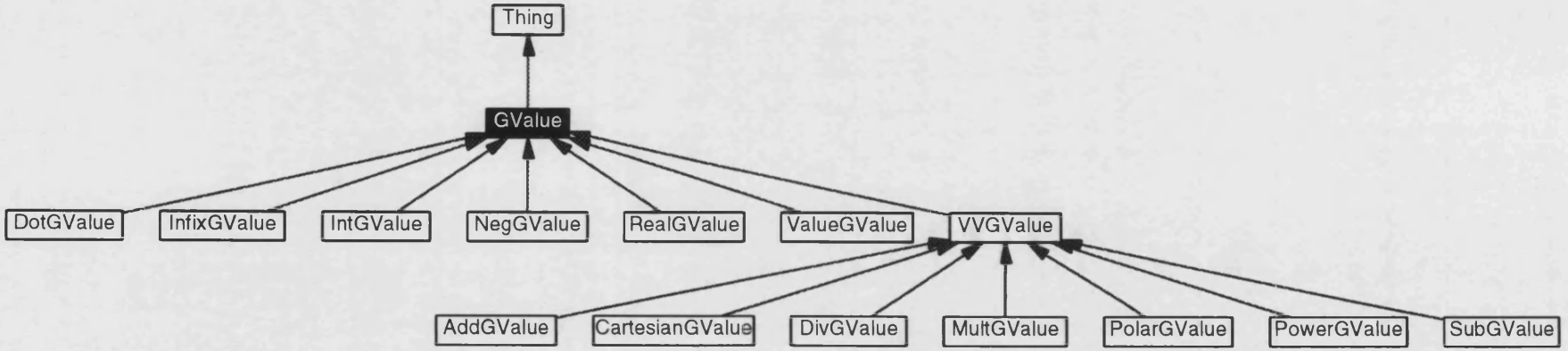


Figure 7.2: Inheritance Diagram For GValue

Public Methods	
	AddGValue (GValue *arg1, GValue *arg2, const char *str)
const char*	**type** ( ) const

Table 7.6: AddGValue Public Interface

only other function implemented is *_reco*, listed as follows:

```
bool
AddGValue::_reco()
{
  ThingListIter iter(parentList());
  XValue arg1 = ((GValue *)iter.next())->xvalue();
  XValue arg2 = ((GValue *)iter.next())->xvalue();
  _xval.add(arg1, arg2);
  return true;
}
```

The *_reco* function reads the list of parents, it knows it depends upon two *GValue* objects as so casts the pointers of the *Things* to *GValues* which it then reads the *XValue* representations from. The *XValue* representation it owns is then used to add the representations, storing the answer in its own representation. The order of the parent list is always maintained, chapter 3, so that this casting can be performed safely without any need to check the identity of the parents. This casting will only identify the *GValue* interface of the parent objects, the identity of the concrete representation, this could be another *AddGValue* or an *InfixGValue*, is never known. Hence the addition of other value concrete representations can be seen, not only to be independent of existing implementation, but also quite straightforward with only a constructor and the *_reco* function to write. For values, the constructor simply sets the object's name and attaches the object to the child lists of the parents. For binary value objects, those operating upon two values, this is done by an additional layer they inherit that sits between them and the *GValue*, this makes implementation of binary values ever easier, by just a small amount.

```
AddGValue::AddGValue(GValue *arg1, GValue *arg2, const char *str)
  : VVGValue(arg1, arg2, str)
{
  // reco ensures the object is updated after initial construction,
  // this isn't really necessary.
  reco();
}

VVGValue::VVGValue(GValue *arg1, GValue *arg2, const char *str)
  : GValue(str)
{
  // VVGValues are attached to their parent's child lists here..
  adoptParent(arg1);
  adoptParent(arg2);
}
```

Other value objects for use by the designer are as follows. Any value referenced on the right hand side may be the reference to a named value, such as x or radius, a numeric value, such as 1.23 or 3.4e-7, or a nested definition, such as mult(2, radius), 2 * radius, or indeed any of the value types listed below:

AddGValue : addition of two values
Syntax: value_variable = add(first_value, second_value)

CartesianGValue : defines complex numbers as cartesian values
Syntax: value_variable = cartesian(real_value, imaginary_value)

DivGValue : division of two values
Syntax: value_variable = div(a_value, divide_by_value)

DotGValue : dot product of two vectors
Syntax: value_variable = dot(first_point, second_point)

InfixGValue : solves mathematical expression with the infix syntax
Syntax: value_variable = a_value + 2 / (another_value - 2)...
Understood unary operators are "+" and "-",
understood binary operators are "+", "-", "*", "/" and "^" (power).

IntGValue : integer value

Syntax: value_variable = int(5)

Shortcuts: value_variable = 5, +/-5, (5), (+/-5)

MultGValue : multiplication of two values

Syntax: value_variable = mult(first_value, second_value)

NegGValue : negates a value variable

Syntax: value_variable = neg(a_value)

Shortcuts: value_variable = -a, (-a)

PolarGValue : defines complex numbers as polar values

Syntax: value_variable = polar(magnitude_value, angle_value)

PowerGValue : value to the power of another value

Syntax: value_variable = power(a_value, to_power_of_value)

RealGValue : floating point value

Syntax: value_variable = real(5.5)

Shortcuts: value_variable = 5.5, +/-5.5, (5.5), (+/-5.5), +5.5e7 -5.5e-7...

SubGValue : subtraction of two values

Syntax: value_variable = sub(a_value, subtract_this_value)

ValueGValue : assignment

Syntax: value_variable = value(a_value)

Shortcuts: value_variable = a, (a)

This value will allow one value to follow the value of another.

7.3 Geometric Building Blocks

Graphically there is no representation for a value object. All the objects subsequently explained can be visualised and so utilise a *Geometric Thing* layer of specialisation, through inheritance, with this layer being derived from the *Thing* base defining the communicative properties. This layer defines additional functions for use in the display and interaction of geometric properties. The *displayable* function, illustrated in table 7.7, can be implemented to return a pointer used to display this object if it can be represented graphically. If this is the case,

Public Methods	
virtual GeometricThing*	**displayable** ( )
virtual unsigned	**displayLayer** ( ) const
virtual void	**displayLabel** (GDisplay &display)
virtual void	**displayObject** (GDisplay &display)
virtual bool	**selectable** ( ) const
bool	**selected** ( ) const
void	**setSelected** (bool state)
virtual XPoint	**centre** ( )
virtual XBox	**boundingBox** ( )
virtual double	**proximity** (const XPoint &seed)

Table 7.7: GeometricThing Public Interface

displayLayer describes a layer which allows the display to view only this type of object, in addition to other specific layers, which simplifies the model view. Reference to the display is then given to *displayLabel* and *displayObject* functions so that these geometric objects can tell a display what to draw, such as draw a point here and a segment there, without actually having knowledge of the display implementation. This collaboration means that geometric objects do not have detailed knowledge of display implementation or of any particular displays and the lifetime any knowledge is just for the duration of these functions. This keeps the geometric objects simple and allows the use of different displays; multiple displays can be used to view different aspects of the model, perhaps focusing on different areas of a machine, and different display implementations based on the generic display can be used. Currently there exist two displays, one capable of on screen representation and the other capable of the postscript output used to illustrate objects within this writing. A display knows how to draw points and segments, all objects being representable through these, so the geometric object knows nothing of lower level lines and arcs which might change from one display type to another.

The other functions within the *GeometricThing* interface deal with the selection or detection of geometric objects. A point and click interface uses the *proximity* function to determine the nearest object, that object can then be marked as selected using an additional flag along the same lines as the *pending* and *invalid* flags implemented within the *Thing*.

7.4 Points

The point objects start to lay the foundations for the node and element representation of the electrical machine by containing a node which will be referenced by dependent objects. The layers of objects that will build on these points will implement additional nodes with these nodes being the corner stones in their design. Nodes are implemented at this level because the equations describing the sought properties of the electrical machine are associated with these nodes. This allows properties to be bound to the building block objects, such as boundary properties, which will then be associated with these nodes and their equations. Later, post-processing can be done on the machine by reading in answers using the nodes to associate equations back with the original objects. Post-processing is future work, see chapter 9.

7.4.1 The Beginnings of Nodal Management

GPoint's are not constructible objects, only instances of the concrete representations may be built. The protected interface of table 7.9 illustrates this by allowing constructor access to derived classes of the object only. This constructor takes, and stores in the generic *GPoint* of table 7.10, the reference to a node builder. When a point is first built it asks the node builder for a new point. The node builder maintains references to all points it allocates, this allows quick identification of all nodes through the node builder. Every time a node aware object uses this point and thus its node, the node builder will increment an index of the number of uses this node has; this allows good management of nodes because the index should zero if all objects have relinquished their reference to the node. If this is not the case, as is often the case during development of new objects, there's a failure in the node management of an object. Therefore, when this point is deleted it tells the node builder it no longer needs this node; hence the reference to the node builder which will be used in the destructor of this object. This mechanism allows the node builder to ensure nodes are not redistributed to other objects when objects in the undo history still reference them; these objects may come back into play if the designer facilitates the use of the undo buffer. Position of this node will be updated in the *_reco* function of the concrete representation as the position of the point invariably changes.

Public Methods

virtual	**~GPoint** ( )
const XPoint&	**point** ( ) const
const Node*	**node** ( ) const
bool	**selectable** ( ) const
XPoint	**centre** ( )
XBox	**boundingBox** ( )
double	**proximity** (const XPoint &seed)
GeometricThing*	**displayable** ( )
unsigned	**displayLayer** ( ) const
void	**displayLabel** (GDisplay &display)
void	**displayObject** (GDisplay &display)
bool	**canWrite** ( ) const
virtual void	**outX** (ostream &out)
const char*	**family** ( ) const

Table 7.8: GPoint Public Interface

Protected Methods

GPoint (NodeReferenceBuilder *nodeReferenceBuilder,
const char *str)

Table 7.9: GPoint Protected Interface

Protected Attributes

XPoint	_xpnt
NodeReferenceBuilder*	_nodeReferenceBuilder
const Node*	_node

Table 7.10: GPoint Representation

Examination of the generic point interface, table 7.8, shows that this base object mainly implements functions within the *Thing* and *GeometricThing* interfaces which are applicable to all points; these basically allowing state specific information to be output and point objects to be displayed and detected within graphical interfaces. These functions aside, all that's left are interfaces to the point representation, used geometrically, and the node used for nodal construction of the electrical machine. This is a tell tale sign that the node structure is a latter addition to geometric objects, designed in parallel to the geometric system so that the two operate together but quite independently. This becomes even more apparent when nodes are examined and seen to be derivatives of the of the *XPoint*, used as the generic representation within the communicative *GPoint* object. The latter section of this chapter illustrates the design of these reference objects.

7.4.2 Point Representation

Like values, points use separate objects as part of their generic representation. This representation can be passed to the display, telling it to draw the point at its specific location; this simplifies displays because they need no knowledge of the more complex communicative *Thing* derived objects of which the additional information held is of no use to the display. Point representations can also be reused to describe vectors, passed to other independent representations using coordinate definitions, and passed to mapping objects which will translate the coordinates using transformation matrices. Coordinates are so frequently used, the encapsulation of x, y, and z coordinate values is very useful. The number of repetitive operations based on coordinates that need repeating for the different axes have been reduced, in terms of repeated program code, through the inclusion of mathematical operators proved by the representation itself. The *XPoint* representation shown in table 7.11 provides nothing but access to the coordinate representation and mathematical operators to act upon this representation. Where integer and floating point values, of the C++ programming language kind, can be added, subtracted, compared for equality and the lack of, so too can points having had these operators defined with the *operator* syntax shown in the public interface.

A nice diagram illustrates the relationships of the point object, figure 7.3. The solid line between the *GPoint* and *Thing* shows inheritance, the dotted line between *GPoint* and *XPoint* shows object aggregation; the communicative *GPoint*

Public Methods

	XPoint ( )
	XPoint (const XPoint &p)
	XPoint (double x, double y, double z)
double	**x** ( ) const
void	**setX** (double x)
double	**y** ( ) const
void	**setY** (double y)
double	**z** ( ) const
void	**setZ** (double z)
void	**reset** ( )
void	**invert** ( )
XPoint	**operator+** (const XPoint &p) const
XPoint	**operator-** (const XPoint &p) const
XPoint	**operator** * (const double v) const
XPoint	**operator/** (const double v) const
bool	**operator==** (const XPoint &p) const
bool	**operator!=** (const XPoint &p) const
double	**cross** (const XPoint &p) const
double	**dot** (const XPoint &p) const
double	**mag** ( ) const
double	**sqr** ( ) const
void	**normalize** ( )
	set (double x,
void	double y,
	double z)
	set (const XValue &x,
void	const XValue &y,
	const XValue &z)
void	**set** (const XPoint &p)

Protected Attributes

double	_x
double	_y
double	_z

Table 7.11: XPoint Public Interface and Representation

has an *XPoint* representation and, as was explained with values, it could be argued that the *GPoint* is a point and should use inheritance instead of aggregation. However, note the line of inheritance between the *Node* and *XPoint*; the *Node* is actually a point and a reference managed point too, the *Buildable* part, enabling the node specific reference builder management of this object discussed at the end of this chapter. Inheritance can't allow multiple inheritance of the *XPoint* object which the *GPoint* object would require in order to be both a point and a node.

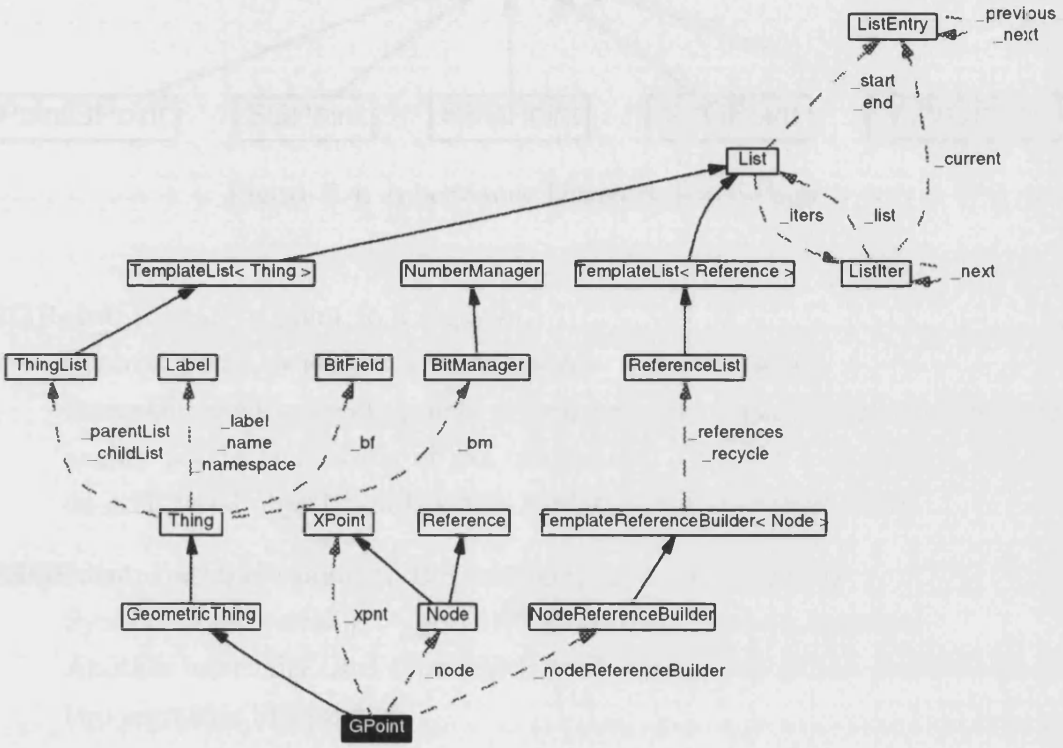


Figure 7.3: Collaboration Diagram For GPoint

7.4.3 Concrete Representations

Now we move onto the building block point objects that the designer can actually construct, illustrated in the inheritance diagram of figure 7.4.

PointGPoint : assignment

Syntax: point_variable = point(a_point)

This allows the value of one point to follow another.

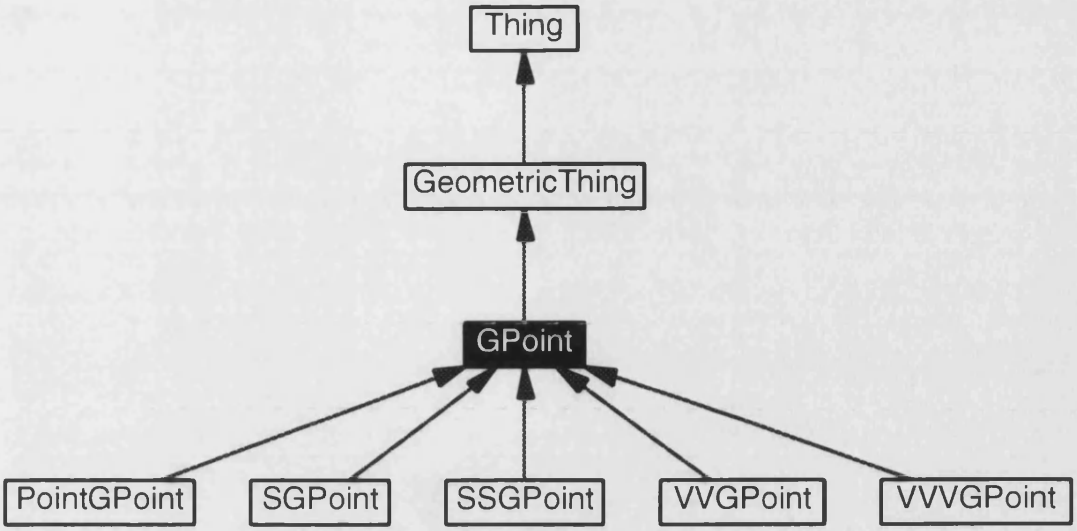


Figure 7.4: Inheritance Diagram For GPoint

SGPoint : attach a point to a segment

Syntax: point_variable = spoint(a_segment, index_value)

Internally used to create points on segments that would otherwise lack the anchor points facilitating object connection. Chapter 6 contains a section on additional object builds which explains this in greater detail.

SSGPoint : attach a point to the intersection of two segments

Syntax: point_variable = sspoint(first_segment, second_segment)

Another internally used type which produces a point at the position where two segments intersect

VVGPoint : x, y coordinate

Syntax: point_variable = vvpoint(x_value, y_value)

This type, and the next, will be used most frequently to provide the anchor point for segments that will form the machine's outline.

VVVGPoint : x, y, z coordinate

Syntax: point_variable = vvvpoint(x_value, y_value, z_value)

The prototypes for *VVGPoints* and *VVVGPoints* really ought to be combined into a simple value dependent point. The point objects themselves could also be combined, it is a trivial task to detect in the *_reco* function whether the object uses two or three values.

7.5 Segments

Segments are one of the most interesting of the building block objects. Historically they originated from what is now the line segment, a line defined by two points. These lines were intersected to produce intersection points, the approach being that the designer didn't need to draw the detailed outline of the machine part by placing all the points by which the lines would be connected. That method would require parameterisation of all points, complicating the design. Figure 7.5 shows that for the slot width to vary, the four width controlling points would need parameterisation to allow them to slide along a tilted axis; if we also want to vary the slot depth then we have to incorporate further parameterisation into the values controlling these points to allow them to slide along another axis. It would certainly seem a daunting task mathematically, and in effort, to produce this parameterisation, there would need to a library encompassing many reuseable examples for this method to become viably useable.

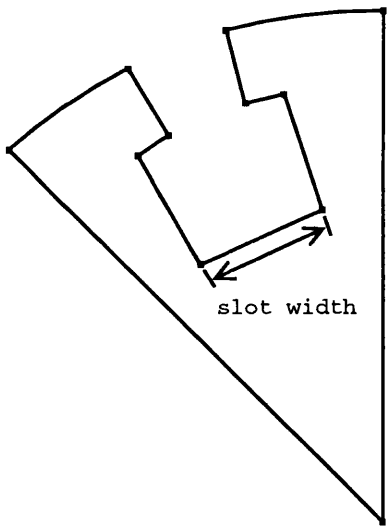


Figure 7.5: Parameterisation Through The Positioning Of Points

The preferred scheme uses construction lines to either form the direct outline, or provide intersection points that will serve to allow subsequent anchorage of construction lines. This method's advantages lie in the fact that we're usually trying to modify some aspect of the electrical machine, governed by parameters measured between lines; the slot depth and width both concern distances between parallel lines in the example of figure 7.5, the angle of the slot, modified to alter the number of slots within a machine, is the angle between two lines. Figure 7.6

shows a pattern of construction lines and arcs used to build a parameterised slot, the central radiating line has been predominantly used to provide anchor points from intersections with the three arcs. Figure 7.7 shows the slot outline with all superfluous construction lines removed and labelling of the adjustable parameters. Parameterisation based on points may still be performed, the example revolves around a centre point upon which everything is dependent, segment intersections simply allow a different design approach more suitable for certain applications.

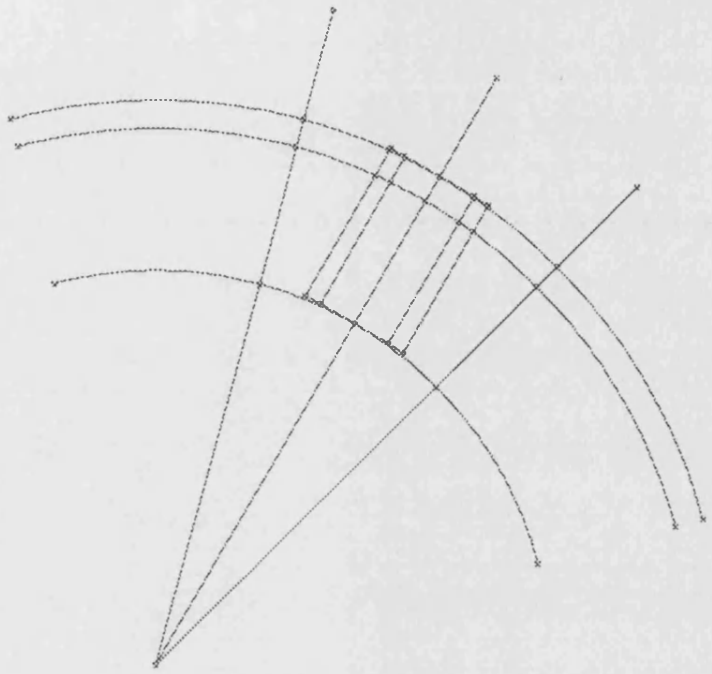


Figure 7.6: Parameterisation Through The Positioning of Segments

7.5.1 Segments As A Parameterised Line

As mentioned earlier, segments originated from a line dependent upon two points. In order to intersect two segments, the quickest method was to define them as a parameterised line. Equations 7.1 and 7.2 describe the x and y equations for parameterised lines respectively, detailed in figure 7.8:

$$x = x_0 + ft \tag{7.1}$$

$$y = y_0 + gt \tag{7.2}$$

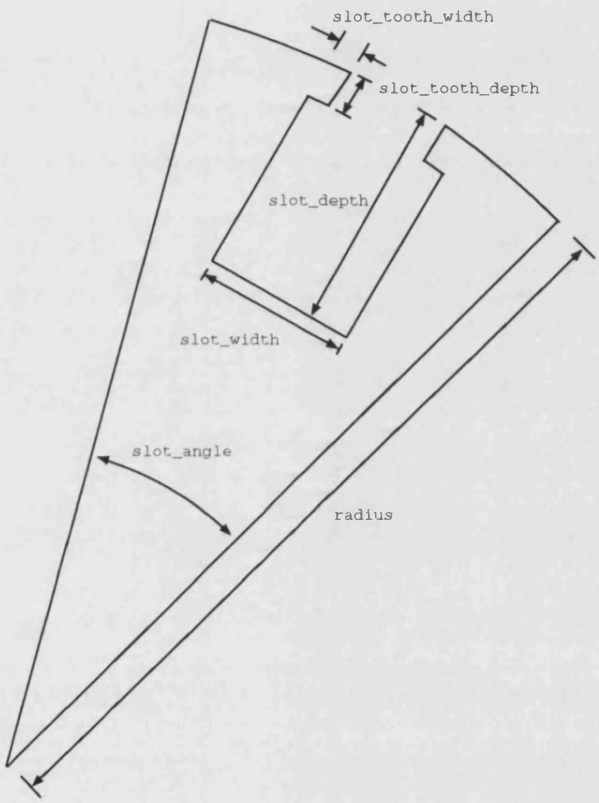


Figure 7.7: Rudiments of a Parameterised Slot

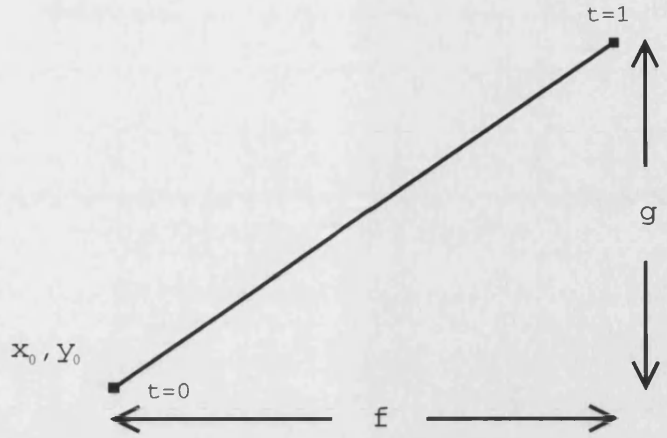


Figure 7.8: Parameterised Line

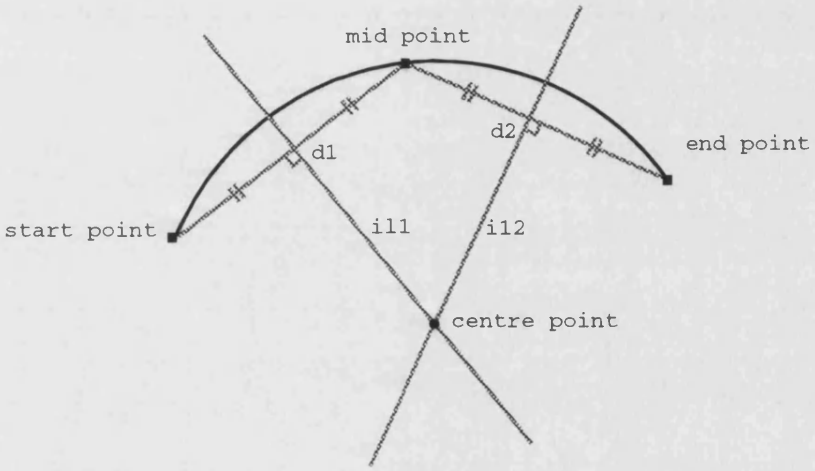


Figure 7.9: Arc Defined Using Three Points

Arcs were then added, the first type created using a similar type to the line segment where three points defined the start, end, and intermediate point as shown in figure 7.9. In order to utilise the arc we had to find the centre and radius, the figure shows how two line segments are drawn between the start point and mid point, the mid point and the end point. Dissecting these two lines into equal parts gives us points *d1* and *d2* from which we can calculate the normals to our original lines, matching their length as an arbitrary length to give the normals. Now the normals are converted into implicit line types governed by equation 7.3.

$$ax + by + c = 0 \tag{7.3}$$

Intersecting the two implicit lines, marked *il1* and *il2* in figure 7.9, gives us the intersection point that is our centre.

7.5.2 Implicit, Parameterised Lines and Segments

With implicit, parameterised and segmented lines having their place, construction lines were initially thought of as being implemented through implicit lines. Figure 7.6 shows that construction lines can have an arbitrary length, so long as they extend outside the bounds of the slot in order to provide the necessary intersections. Implicit lines could later be cropped to tidy the design and produce lines of specific lengths, these being implemented through either parameterised lines or line segments. The process of tying this into a coherent scheme proved difficult; the inclusion of arcs into this scheme complicated matters further, especially if we wanted to interchange lines with arcs. As is often the case, if a design proves to difficult to conceive it's probably taking the wrong approach.

7.5.3 The Abstract Segment

The final scheme uses abstract segments, these having a start and end point with an arbitrary number of points in between. For a line type to be a segment, it must be possible to parameterise it between the values of $t=0$ to $t=1$ according to equations 7.1 and 7.2. Table 7.12 shows the interface functions accessing the *GSegment* object. Like values and points, the constructor is protected signifying that this object is not constructible within its own write; only derived objects, the concrete representations, can be constructed. Also notable are the lack of polymorphic functions within this interface considering its abstract nature. The *XSegment* object held within the *GSegment* representation, table 7.13, is a pointer to another family of objects responsible for the representation side of segments. This family of objects simply concern themselves with the geometric properties of segments, without concerning themselves with the communicative and interactive side the *GSegment* object deals in; in fact, of all the functions within the *GSegment* interface, most functions implement aspects of the communicative *Thing* object and the geometrically interactive *GeometricThing* object, whilst the others provide access to like named functions within the interface of

Public Methods	
virtual	**~GSegment** ( )
XSegment&	**xsegment** ( )
XSegment*	**copy** (double start_t, double end_t)
const XPoint&	**start** ( ) const
const XPoint&	**end** ( ) const
const XPoint&	**point** (unsigned i) const
unsigned	**nPoints** ( ) const
XPoint	**pointAt** (double t) const
double	**atPoint** (const XPoint &p) const
bool	**closedSegment** ( ) const
unsigned	**intersectWith** (GSegment *segment)
const SplitXPoint&	**intersectPoint** (unsigned i)
XImplicitLine	**tangent** (const XPoint &seed) const
bool	**selectable** ( ) const
XPoint	**centre** ( )
XBox	**boundingBox** ( )
double	**proximity** (const XPoint &seed)
GeometricThing*	**displayable** ( )
unsigned	**displayLayer** ( ) const
bool	**canWrite** ( ) const
const char*	**family** ( ) const

Protected Methods
GSegment (const char *str)

Table 7.12: GSegment Interface

the geometric representation.

7.5.4 Segment Representation

Worth noting is the pointer nature of the *XSegment* representation within the *GSegment's* representation of table 7.13. This utilises the polymorphic nature of the representation; the *XSegment* is actually an abstract class through which segment implementations can be accessed. The pointer nature allows any imple-

Protected Attributes	
XSegment*	_xseg
XPoint	_centre
bool	_centre_cached
XBox	_boundingBox
bool	_boundingBox_cached

Table 7.13: GSegment Representation

mentation of the *XSegment* object, created through its inheritance, not only to be plugged into the representation but also to be switched for another representation; this allows an arc to be switched for a line within a *GSegment* implementation that contains both these types, this may happen when three points defining an arc form a straight line.

Looking at the *XSegment* interface of table 7.14, it consists almost entirely of polymorphic functions of the form pure virtual form; in C++ nomenclature [18] [19], this means that an implementation will be looked for in the derived class with there being no implementation in the base class. In other words, the straight line, circle and arc implementations forming the real objects must implement these functions, the program will not compile otherwise. Half these functions access points, points being the representation of a segment; a segment must have two or more points describing a line that can be parameterised. Here's a quick explanation of some functions in the context of their use:

copy :

Given to parameters of t, we can describe any section of the line. Discrete segments identify such a section through the identification of two points lying upon the line. This function allows identification of this section, producing a copy of it that can be used elsewhere.

pointAt :

To obtain the coordinates of a point on the line at a specific parameter of t, *pointAt* returns the coordinate as an *XPoint*. Now we can see the benefit of the point representation as a separate entity to the communicative *GPoint* object.

atPoint :

Public Methods

virtual	**~XSegment** ( )
virtual XSegment*	**copy** (double start_t = 0, double end_t = 0) = 0
virtual const XPoint&	**start** ( ) const = 0
virtual void	**setStart** (const XPoint &s) = 0
virtual const XPoint&	**end** ( ) const = 0
virtual void	**setEnd** (const XPoint &e) = 0
virtual const XPoint&	**point** (int i) const = 0
virtual void	**setPoint** (int i, const XPoint &p) = 0
virtual int	**nPoints** ( ) const = 0
virtual XPoint	**pointAt** (double t) const = 0
virtual double	**atPoint** (const XPoint &p, bool strict = true) const = 0
virtual bool	**closedSegment** ( ) const = 0
virtual int	**intersectWith** (XSegment &segment) = 0
virtual int	**intersectWithCircle** (XCircleSegment &segment) = 0
virtual int	**intersectWithLine** (XLineSegment &segment) = 0
void	**addIntersect** (const XPoint &xp, double t)
void	**clearIntersects** ( )
const SplitXPoint&	**intersectPoint** (int i) const
virtual const XPoint&	**centre** ( ) const = 0
virtual XBox	**boundingBox** ( ) const = 0
virtual double	**proximity** (const XPoint &seed) const = 0
virtual XImplicitLine	**tangent** (const XPoint &seed) const = 0

Protected Methods

XSegment ( )

Table 7.14: XSegment Interface

The reverse to the previous function, *atPoint*, returns the value of t for a given coordinate that, ideally but not strictly, lies on the line.

closedSegment :

A circle is a closed segment, straight lines and arcs are open. Discrete segments divide the section they describe according to a number of nodes, n; if a closed segment is involved the last node overlaps the first and so they can compensate for this by sectioning the length into $n + 1$ nodes, connecting the n^{th} node back to the first.

centre :

Gives the particular segment's idea of what its centre is. This, and the next two functions, help implement the graphical representation and detection of objects used with the *GeometricThing* interface.

boundingBox :

Gives a *XBox* bounding box for the object. The *XBox* implements intersections of bounding boxes allowing detection of an intersection and its union.

proximity :

Returns the distance from the given point to the nearest point on the segment.

tangent :

Given a seed point on the segment, this calculates the tangent to the segment at that point. This proves very useful in the *SPVVVGLineSegment* object explained later, an object that draws a line segment at a given angle to the tangent of a segment.

Remaining intersect functions deal with the problem of intersecting segments through an abstract interface. The implementing functions need some knowledge of the segment's true nature, straight line or circle. If we ask an *XSegment* to intersect with another *XSegment* we ultimately need to intersect between the *XLineSegment* and *XCircleSegment* types, this is because the intersection routines deal with the parameterised, x_0, y_0, g, and f parameters of a line and the centre and radius of a circle. The arcs, *XArcSegment*, are treated as circles. Figure 7.10 shows the arc as a derivative of the circle, it simply implements a start angle and angle of duration. The arc implements some extra functionality onto

the circle to ensure that, in the case of intersections, the resulting intersection of a segment with a circle actually lies on the portion of the circle defined by the arc.

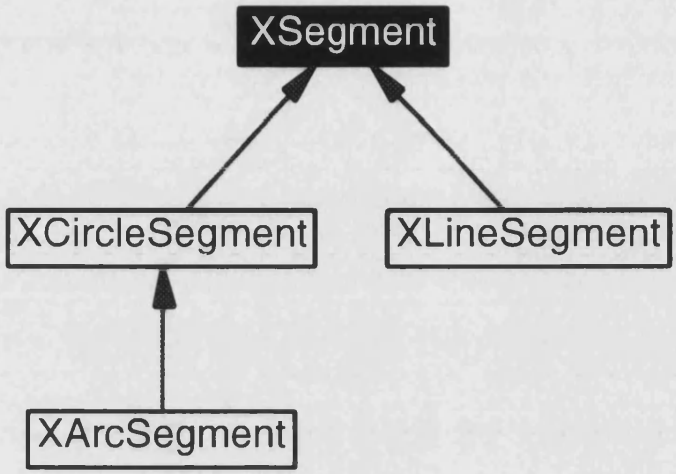


Figure 7.10: Inheritance Diagram For XSegment

Each segment type knows how to intersect itself with a line or circle segment, hence the *intersectWithLine* and *intersectWithCircle* functions in the *XSegment* interface. If we wanted to intersect two segments, l being an *XLineSegment* and c being an *XCircleSegment*, we could do so by asking either segment to *intersectWith* the other. Asking the line l to intersect with the circle we would use:

```
l->intersectWith(c)
```

This passes the *XSegment* interface of c to the line object. Unable to do anything with this abstract *XSegment* description of c, the line asks c to intersect itself with a line segment:

```
c->intersectWithLine(this)
```

The line, quite aware of its own identity, can remove ambiguity by telling the circle, which it doesn't know the identity of, to intersect with a line, passing itself, *this*, as an argument. The circle now performs an intersection with a line[22], storing any resultant intersection points in a cache of points held both

within itself and the line. The initial intersection asked the line to intersect with the circle, ultimately the circle intersected with the line. The results are hence stored in both objects to avoid any confusion. The circle returns the number of intersections to the line with the *intersectWithLine* return value, this allows the line to return this number through the return value of the *intersectWith* function. Whatever initiated this intersection can then obtain the intersection points using *intersectPoint*, the caching mechanism being the only implementation held within the *XSegment* object. The return type of this function, a *SplitXPoint*, gives a point coordinate and the parameter of t at which the intersection occurred; this parameter is thus specific to the intersected object as the line and the circle in our example would intersect at points differing in their parameterisation.

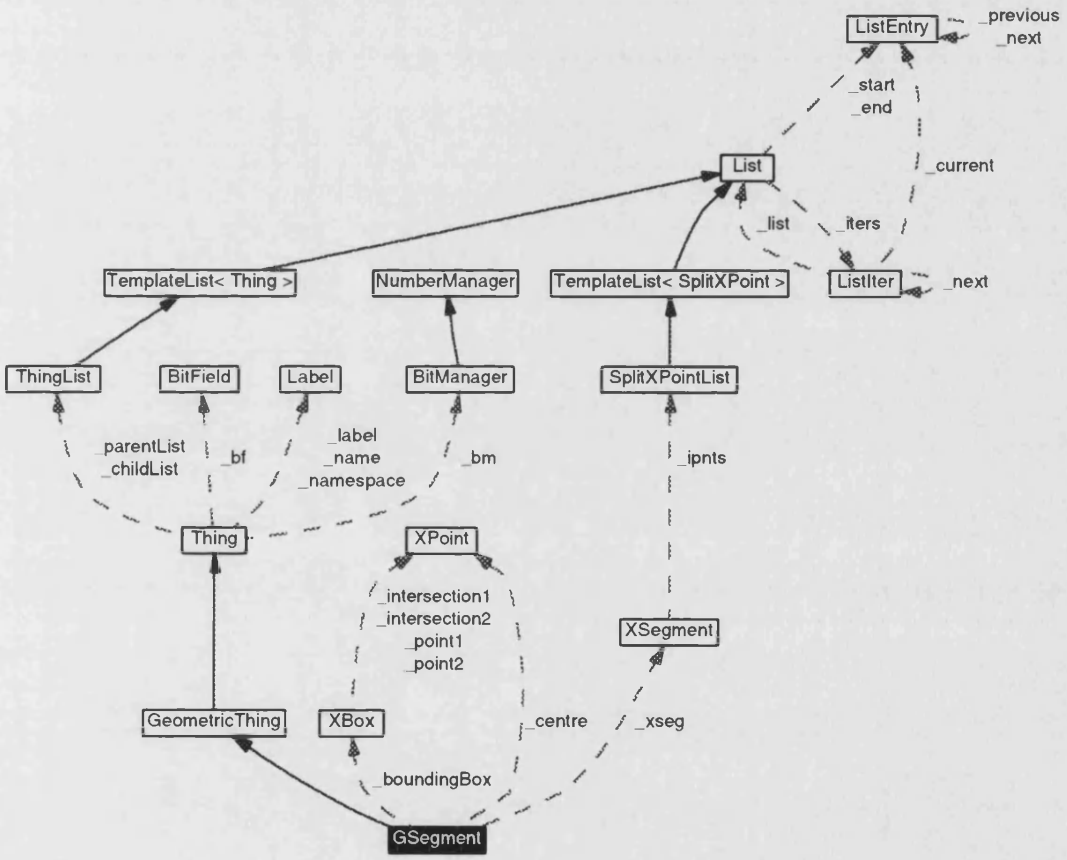


Figure 7.11: Collaboration Diagram For GSegment

Figure 7.11 shows the collaboration between the *GSegment* and *XSegment* objects. The *XSegment* contains a list of intersection points, its only data, the dotted line showing ownership. Straight lines denote inheritance, the *GSegment* being a *GeometricThing* as a specialisation of a *Thing*.

7.5.5 Concrete Representations

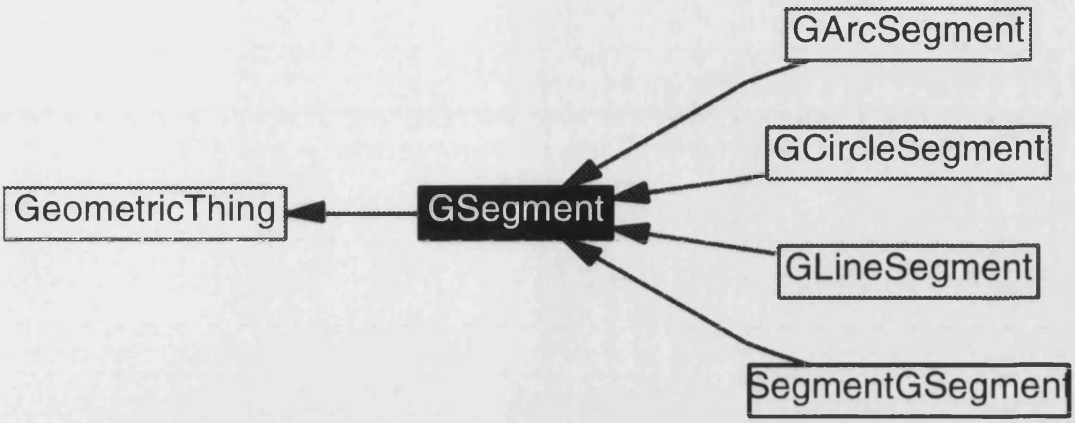


Figure 7.12: Inheritance Diagram For GSegment

The number of objects useable to the designer is of the largest with respect to the segment family. Most of them fall into three main groups due to their type of representation, however all are interchangeable as this representation is hidden behind the *GSegment* interface. The outside world sees a line type which can be parameterised, with non-closing line types, such as arcs and straight lines, having terminating points with which to aid subsequent anchorage of objects. Figure 7.12 shows the three families of *GArcSegment*, *GCircleSegment*, and *GLineSegment*, in addition to the *SegmentGSegment*, an assignment type that exists independently. This type facilitates the importing of library components by allowing an imported segment to depend upon an existing segment within the model, taking a copy of its representation.

Arc Segments

The first of the three families uses an *XArcSegment* representation. All concrete representations within this family convert their defining parameters into the default representation of three points, a start point, intermediate point, and end point, through which the arc passes defining its path. The *GArcSegment* family currently implements three concrete types, shown in figure 7.13.

PPPGArcSegment : figure 7.14

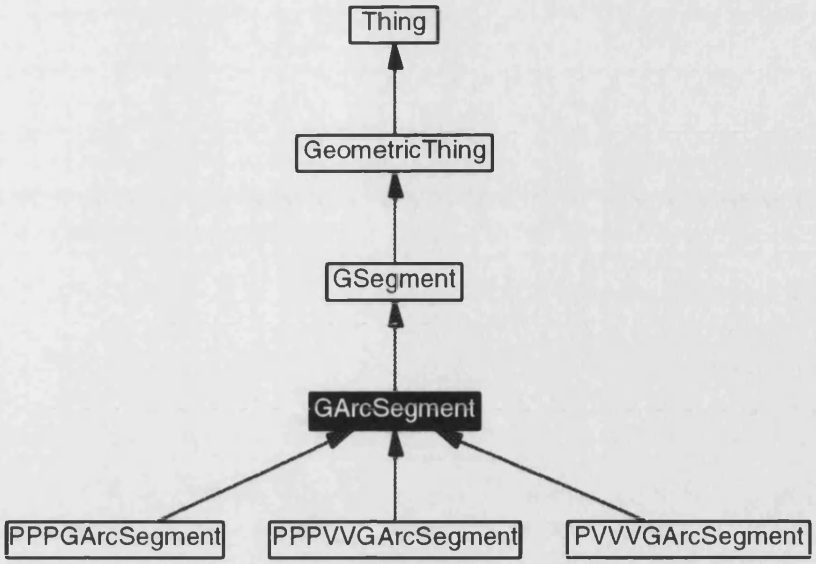


Figure 7.13: Inheritance Diagram For GArcSegment

Syntax: segment_variable = ppparc(start_point, mid_point, end_point)

Dependent upon three points, this arc will find the path that takes it from the start point, through the mid point and up to the end point. The points defining the arc thus define its anchor points for subsequent attachment.

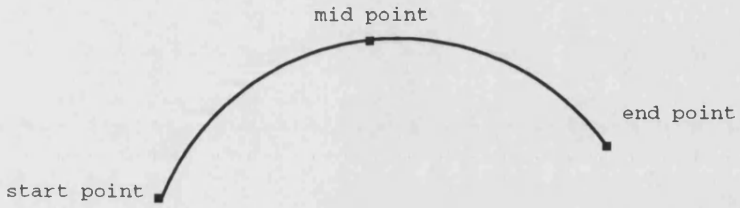


Figure 7.14: Arc Dependent Upon Three Points

PPPVVGArcSegment : figure 7.15

Syntax: segment_variable = pppvvarc(start_point, mid_point, end_point,
length_extension_from_start_point,
length_extension_from_end_point)

Dependent upon three points and two length values. Being similar to the previous *PPPGArcSegment*, this type creates two new points extending

along the path of the arc from the defining start and end points. This allows the arc to project outwards providing new anchor points, possibly intersecting neighbouring segments. Two new points are created at these new extents, these being of the *SGPoint* variety dependent upon the arc itself.

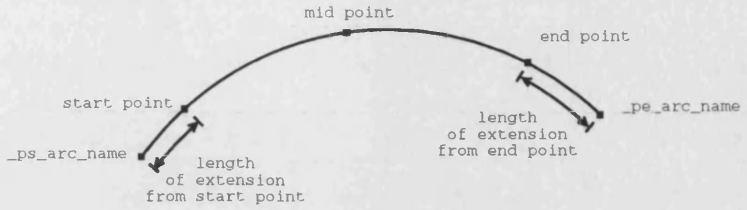


Figure 7.15: Arc Dependent Upon Three Points and Two Extension Values

PVVVGArcSegment : figure 7.16

Syntax: segment_variable = pvvvarc(centre_point, radius_value,
start_angle_value, through_angle_value)

This type depends upon a centre, radius and two angles. One of the most commonly used methods for arc representation, useful for defining the foremost construction lines where the radius matches that of the machine. The only point used to define this arc is its centre, therefore, in order to fulfil the specification that an arc must supply anchor points for subsequent connection, this arc produces to dependent point at the extremities of its path.

Circle Segments

The family of *GCircleSegments* uses an *XCircleSegment* representation to provide concrete representations useful for founding construction lines within the machine and the definition of holes in slots. It's representation is the same as the arc's, the arc being a specialisation to the circle inheritance of figure 7.10. This segment type is closed, with no start or end to the segment. Unless the concrete representation uses points to define the path of the circle, no points will exist

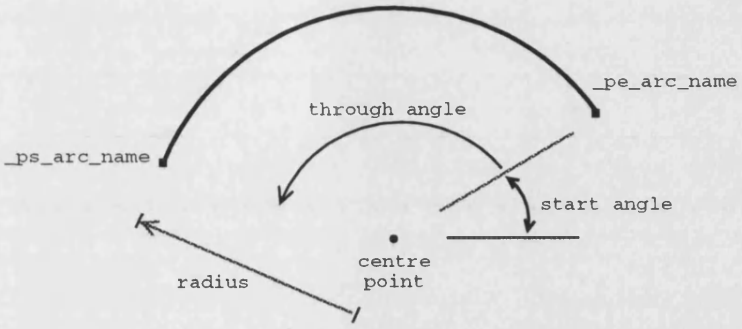


Figure 7.16: Arc Dependent Upon Centre Point, Radius, Starting, and Through Angle Values

on the circle for subsequent anchorage. The intersection of the circle with other segments produces intersection points facilitating this. Only two concrete types exist, shown in figure 7.17.

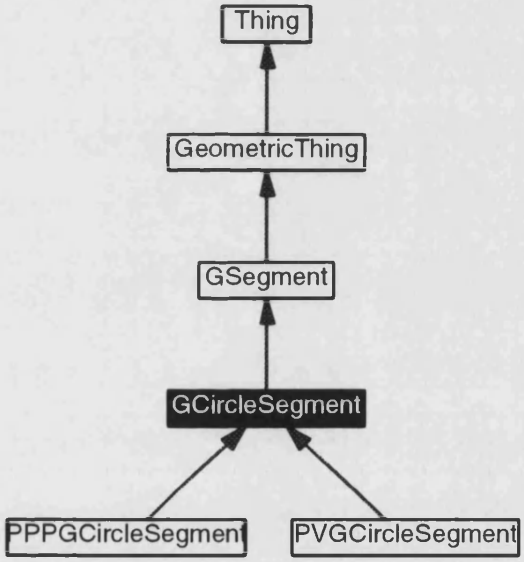


Figure 7.17: Inheritance Diagram For GCircleSegment

PPPGCircleSegment : figure 7.18

Syntax: segment_variable = pppcircle(start_point, mid_point, end_point)

Dependent upon three points, this circle will find the circular path that intersects all these points.

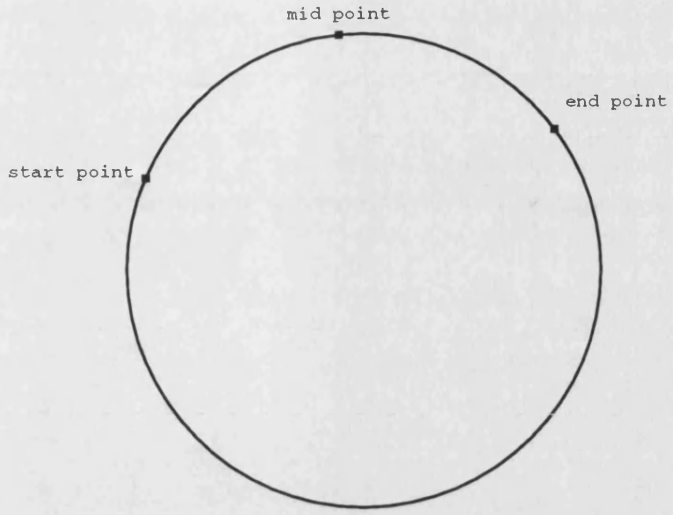


Figure 7.18: Circle Dependent Upon Three Points

PVGCircleSegment : figure 7.19

Syntax: segment_variable = pvcircle(centre_point, radius_value)

Dependent upon a centre point and radius value. Placement of a few of these circles around the machine's centre will define the machine's radius and useful intersection points, between lines radiating from the centre, for dependencies upon slot depth and other depth specific parameters.

Line Segments

Here we have the largest family of segments, using the *XLineSegment* representation consisting of two points connecting the line. The available types of figure 7.20 are briefly describe as follows:

FPGLineSegment : figure 7.21

Syntax: segment_variable = ppline(start_point, end_point)

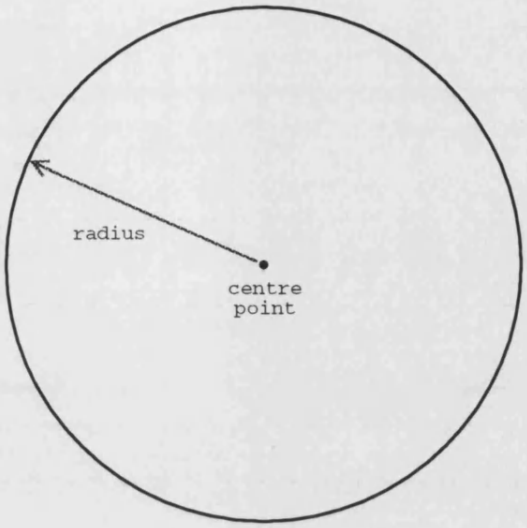


Figure 7.19: Circle Dependent Upon A Centre Point and Radius Value

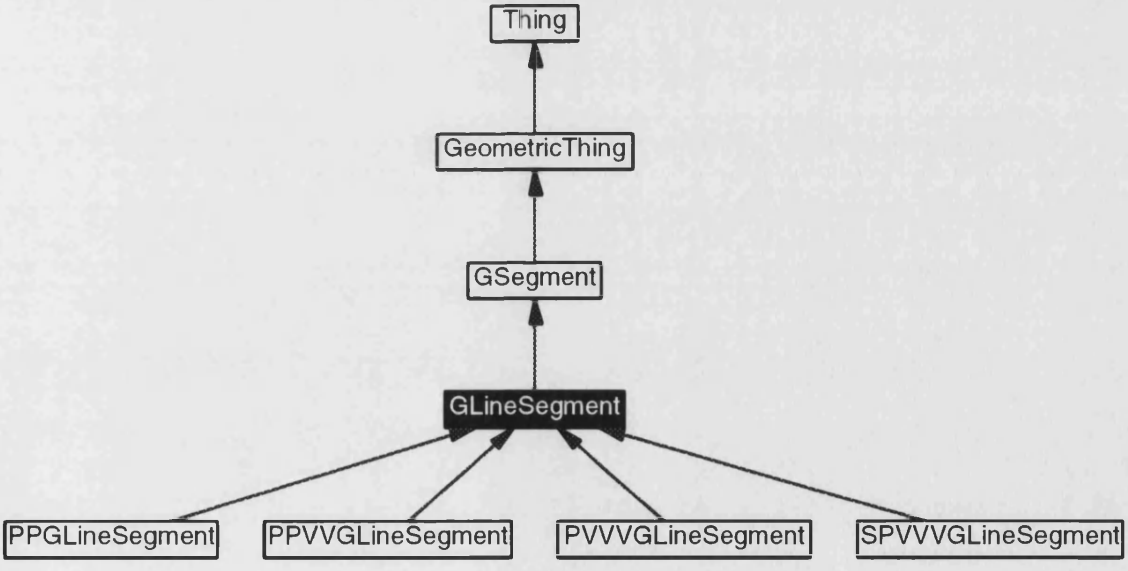


Figure 7.20: Inheritance Diagram For GLineSegment

Dependent upon two points, the points defining this line also provide the necessary anchor points.

Figure 7.21: Line Dependent Upon Two Points

PPVVGLineSegment : figure 7.22

Syntax: segment_variable = ppvvline(start_point, end_point,
length_extension_from_start_point,
length_extension_from_end_point)

This types relation to the *PPGLineSegment* resembles the relationship of the *PPPVVGArcSegment* type to the *PPPGArcSegment*. The length of the line can be extended along its path, beyond the extremities defined by the two points it depends upon. This is predominantly used to extend an existing line to the point of intersection with other segment, those points then allowing definition of discrete segments to sub-divide an are for finer meshing.

Figure 7.22: Line Dependent Upon Two Points and Two Extension Values

PVVVGLineSegment : figure 7.23

Syntax: segment_variable = ppvvline(origin_point, start_length, extension_length, angle_of_trajectory)

Dependent upon a point of origin, this line extends at a given trajectory, relative to the 3 o'clock position. The physical line starts at the given length from the origin, extending then by the length of extension. Anchor points are created at the termination of the physical part of the line governed by the start and extension lengths.

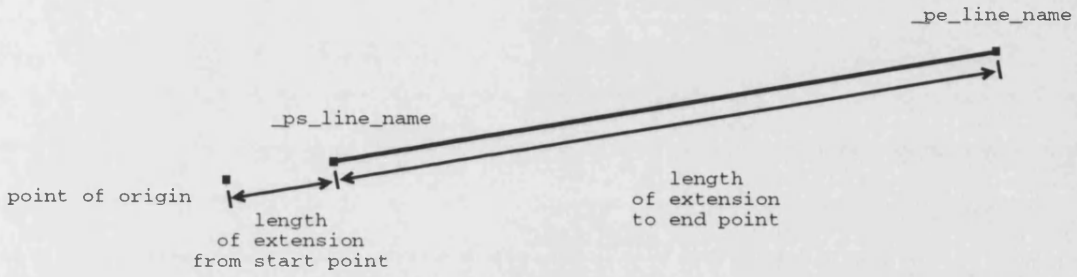


Figure 7.23: Line Dependent Upon a Point of Origin, Start Length, Extension Length, and Angle of Trajectory

SPVVVGLineSegment : figure 7.24

Syntax: segment_variable = sppvvline(segment, point, length_extension_to_start_point, length_extension_to_end_point, angle_relative_to_tangent)

Produces a line segment at an angle to the tangent of a segment at a given point. The line segment extends from this tangential point by given values of extension for the start and end point. These points are manufactured to provide the necessary anchorage.

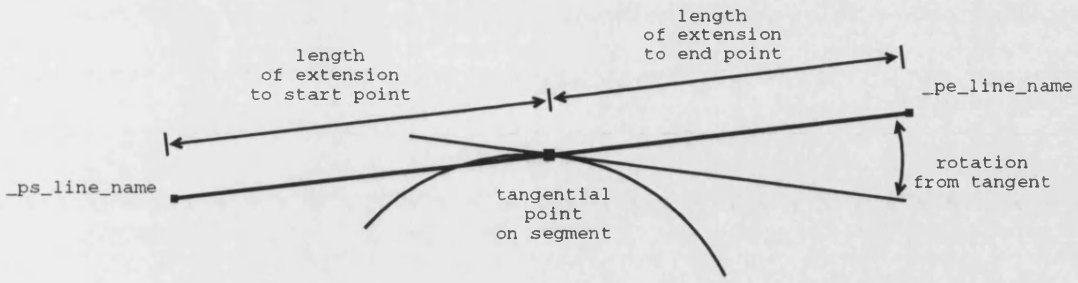


Figure 7.24: Line Dependent Upon Segment's Tangent at the Given Point

7.6 Discrete Segments

A discrete segment identifies a section of a segment, bounded by two points. It inherits the nodes of these two points and then divides the spanned section into $n-1$ smaller sections according to a discretisation policy. Nodes are then created, marking these divisions so that the final discrete segment contains n nodes. A chain of discrete segments marks boundaries for meshing, the density of nodes along this boundary dictating the mesh density. A variation on this type allows the grouping of discrete segments, this allows several segments to appear as one. Objects dependent on specific numbers of discrete segments can thus span a greater number of segments using this type.

This object family is the first explained so far within this chapter not to have a separate generic representation. All it essentially contains, see table 7.16, is an array of node references and a reference to the node builder, used to reference and manufacture nodes. The generic interface to this object is very simple, table 7.15 shows access functions used to read points, used geometrically within the program, and nodes, used in the parallel node reference system. All other functions implement its interaction, based in the implementation of functions within the *Thing* and *GeometricThing* interfaces. The *GDSegment* isn't a constructible object, two concrete representations currently exist which inherit this interface; hence the protected, accessible only to derived objects, nature of this object's constructing interface of table 7.15. These two concrete objects differ significantly in their representation. With the simplicity of one type and the complexity of the other, no separate representation has been used.

Public Methods

virtual	**~GDSegment** ( )
unsigned	**nPoints** ( ) const
XPoint	**point** (unsigned n) const
unsigned	**nNodes** ( ) const
const Node*	**node** (unsigned n) const
bool	**selectable** ( ) const
GeometricThing*	**displayable** ( )
unsigned	**displayLayer** ( ) const
void	**displayLabel** (GDisplay &display)
void	**displayObject** (GDisplay &display)
bool	**canWrite** ( ) const
const char*	**family** ( ) const

Protected Methods

GDSegment (NodeReferenceBuilder *nodeReferenceBuilder,
const char *str)

Table 7.15: GDSegment Interface

Protected Attributes

NodeReferenceBuilder*	_nodeReferenceBuilder
NodeArray	_nodes
XPoint	_centre
bool	_centre_cached
XBox	_boundingBox
bool	_boundingBox_cached

Table 7.16: GDSegment Representation

7.6.1 The Discrete Type

The *SGDSegment* is the main type used, initially being the sole representative of this family. The need for a collective type cause abstraction of the *GDSegment* interface with this type being sub-classed as one of two concrete types. It therefore has a very simply inheritance structure, shows in figure 7.25.

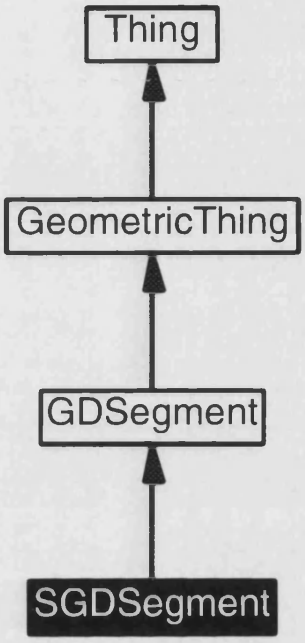


Figure 7.25: Inheritance Diagram For SGDSegment

The following, figure 7.26, illustrates the conversion of a segment into a discrete segment using a linear node spacing. Placement of the nodes is done using the segments ability to be parameterised from $t = 0$ to $t = 1$. For arcs and lines this is simply performed by taking the two parameters of t the segment returns when asked at what parameters the bounding points lie, using the segment's *atPoint* function. The discretisation policy is told how many points are required, it then governs how this difference is divided. Having calculated the values of t where the intermediate points lie, all that remains is the conversion of these parameters back into coordinates. This is performed using the segment's *pointAt* function. The only problems arise when we're dealing with closed segments, namely circles. Here the parameter t identifies the same point for $t = 0$ and $t = 1$. In this case, regardless of their values, if the two bounding parameters of t are equal, we assume that we want to travel the full length of the closed segment. The other problem lies with the direction taken around the closed

segment. If the bounding parameters were 0.75 and 0.25 respectively, do we increase t positively from $t = 0.75$, crossing the $t = 1.0/0.0$ boundary until $t = 0.25$, or do we decrease t negatively from $t = 0.75$ to $t = 0.25$? To solve this we introduce another assumption that we always travel counter-clockwise along the closed segment with t increasing positively in this direction.

If we were to take the slot of figure 7.6, creating discrete segments along the outline of the slot, figure 7.27 would show the discrete segments displayed against the original segments. The next illustration, figure 7.28 removes the segments from the display leaving just the discrete segments. The electrical machine is starting to take form. Each discrete segment's node count is multiplied by a factor of n, allowing, ultimately, the mesh density to be tuned using this parameter.

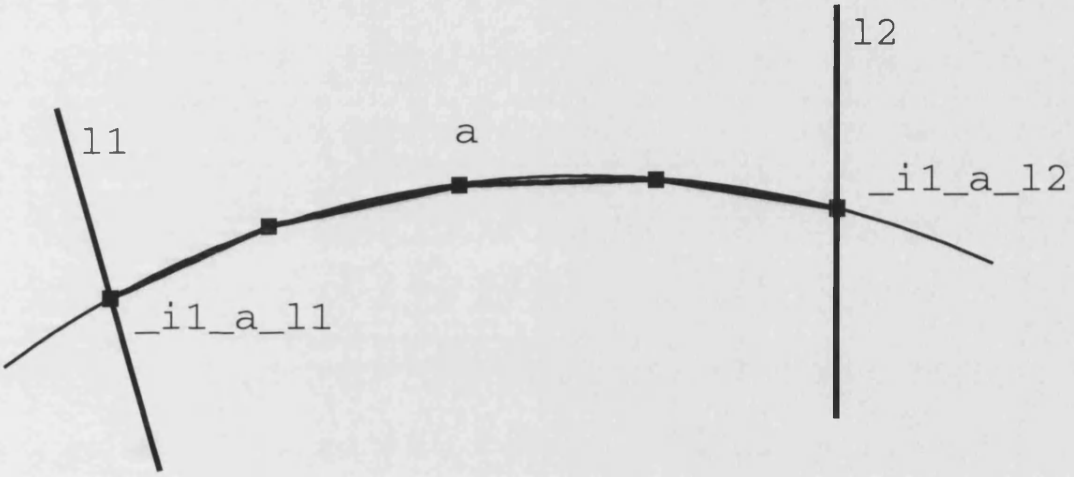


Figure 7.26: Discrete Segment Based On An Arc Segment

7.6.2 Composite Discrete Segments

This form of discrete segment exists to masquerade two or more discrete segments as one. Take the example of figure 7.29, the lower part of the figure forms a triangular section of which the top, horizontal, line is divided into three discrete segments. For a meshing object, consisting of a finite discrete segment interface, to mesh this, these three segments need to appear as one segment. This is not just to overcome the interface to the meshing object. We could overlay the three discrete segments with an additional discrete segment, however this would cause two problems. First, the new segment would create intermediate nodes that would

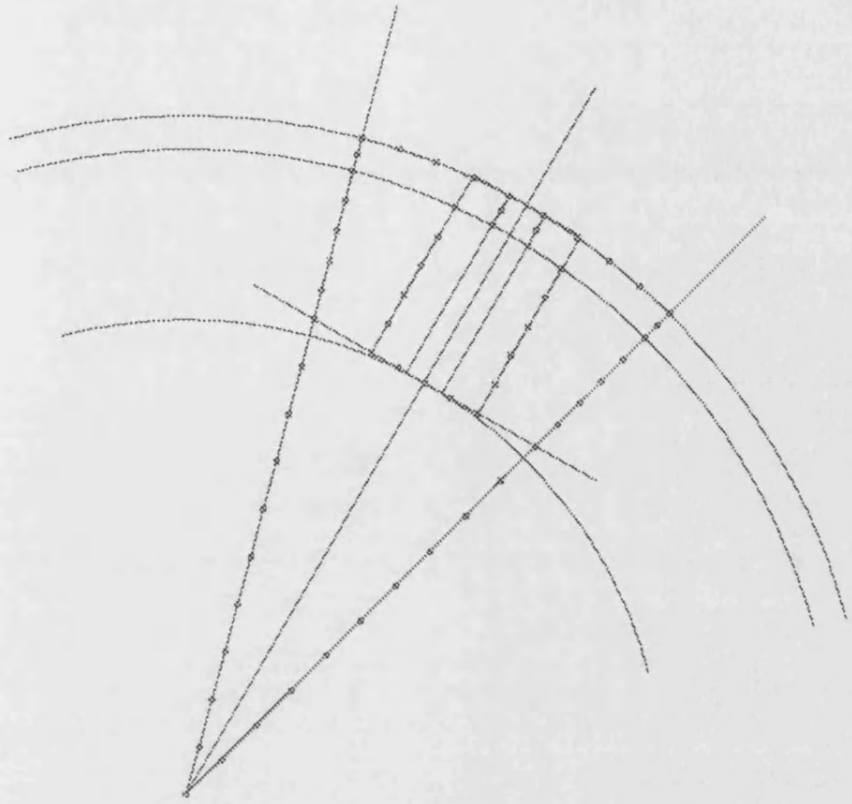


Figure 7.27: Discrete Segments Based On A Slot Example, Segments Shown

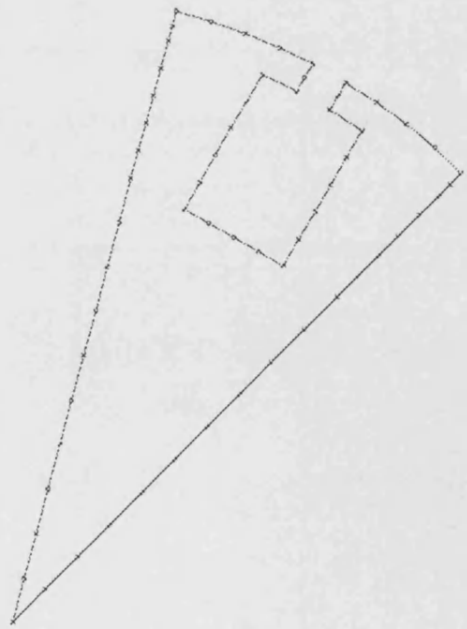


Figure 7.28: Discrete Segments Based On A Slot Example, Segments Removed

affect our node referencing by producing duplicate nodes at the same points in space as the nodes of the other segments. Secondly, if the number of nodes in either segment were changed so they no longer overlaid one another, the nodes on the boundaries of the adjacent meshes would not correspond and we would require a solver capable of overcoming this. By producing a composite discrete segment of the form "ds_21_8_20 = cdsegment(ds21, ds8, ds20" we would avoid this, the new discrete segment would reference the other segments nodes thus mimicking their behaviour.

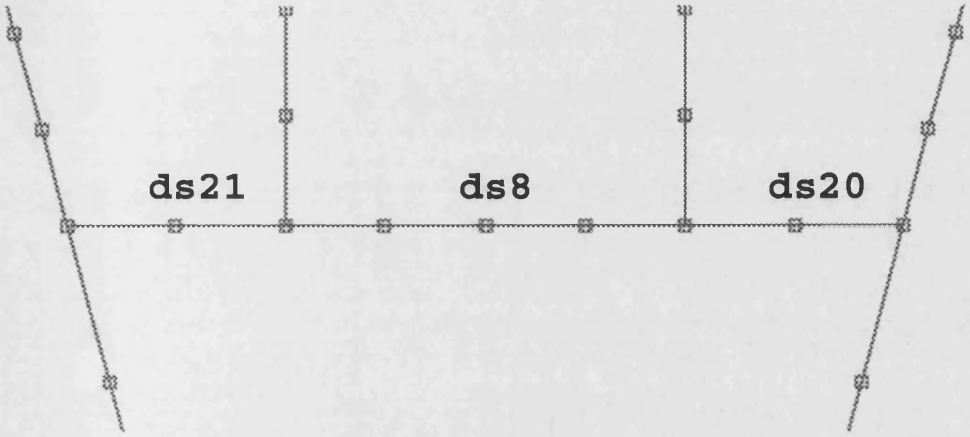


Figure 7.29: Composition of Discrete Segments

Chaining Discrete Segments

Masquerading discrete segments as one involves the sorting of each individual discrete segment into a continuous path. Adjoining discrete segments will reference a common node, helping in the search, these nodes can be overlaid resulting in a continuous chain of nodes. Some chains will close, others will remain open; in order to ensure the best connectivity, the node furthest from all other nodes denotes the start of an open chain. If no exists, takes the c
nodes either side of the discontinuity are not overlaid; this effectively inserts an extra link in the chain, this way we only every have the one, largest, discontinuity in any open chain. Use of a discrete segment organiser is quite prevalent in the following objects, it has thus been created as a reuseable object which the composite discrete segment, *CGDSegment*, inherits. Figure 7.30 illustrates this. The *CGDSegment* inherits the *GDSegmentChain* object as it is indeed a chain of segments, also it will never contain more than one such chain, a candidate for

the use of object aggregation.

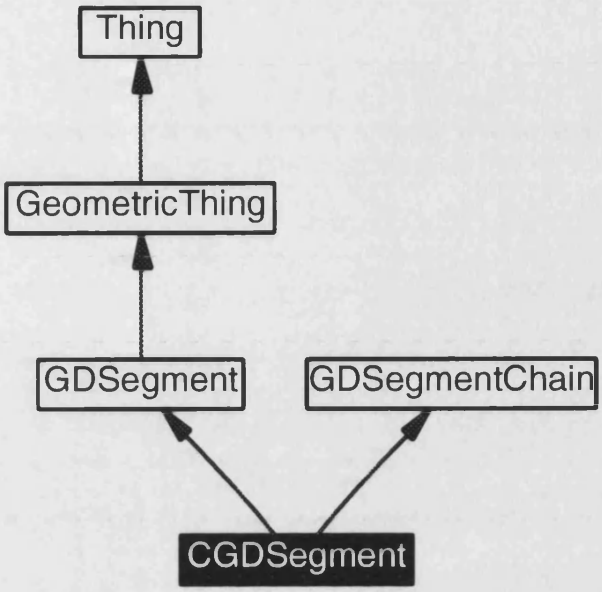


Figure 7.30: Inheritance Diagram For CGDSegment

7.7 Fronts

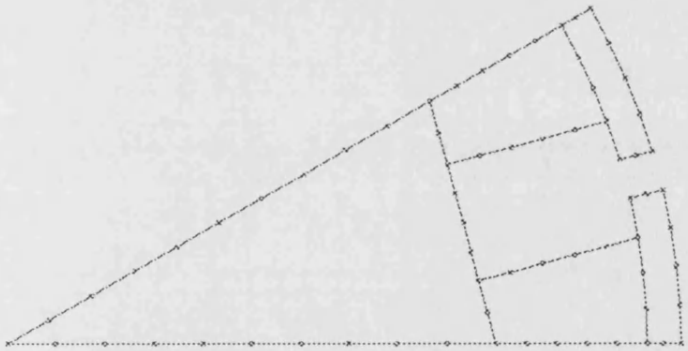


Figure 7.31: Slot Sub Division Into Five Domains

In meshing an area we have the choice of either dividing that into manageable domains, figure 7.31 shows the slot example with additional discrete segments added, or of treating it as one domain, figure 7.32. The foremost method allows greater control over the mesh, the domains are simple three or four sided areas which can be meshed quite precisely, whilst the latter method relies on a meshing method capable of handling the more complex geometry. Such a method very

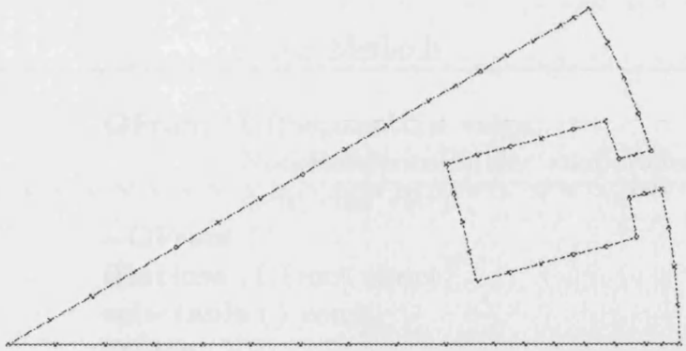


Figure 7.32: Slot Treated As A Single Domain

much puts us at the mercy of the mesh generating software we have available; many mesh generators are available within the public domain, these predominantly use methods based on the Delauney Voronoi method[1][2], producing triangular elements. If quadrilateral elements are preferred, these will not bear the desired results. They are however capable of handling holes within the meshed domain. To allow the use of these schemes, fronts must be used; a front is capable of grouping a chain of discrete segments into one object, a versatile way of dealing with many segments when interfacing to the more geometrically capable meshing methods. Interfaces to such methods are then capable of differentiating between the different groupings of discrete segments as boundaries, allowing the specification of holes within the mesh.

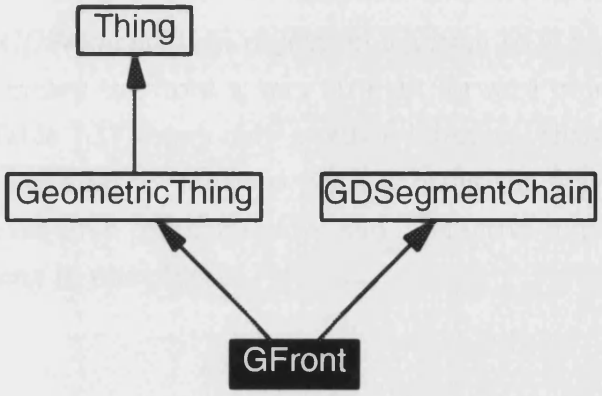


Figure 7.33: Inheritance Diagram For GFront

We have a very convenient way of producing the front object, figure 7.33 illustrates the reuse of the *GDSegmentChain* object. A front can now be specified in terms of an arbitrary list of discrete segments, the *GDSegmentChain* will order

	GFront (GDSegmentList *segs, NodeReferenceBuilder *nodeReferenceBuilder, const char *str)
	~GFront ( )
int	**iEnclose** (GFront *front)
bool	**selectable** ( ) const
XPoint	**centre** ( )
XBox	**boundingBox** ( )
double	**proximity** (const XPoint &seed)
GeometricThing*	**displayable** ( )
unsigned	**displayLayer** ( ) const
void	**displayLabel** (GDisplay &display)
void	**displayObject** (GDisplay &display)
bool	**canWrite** ( ) const
void	**outX** (ostream &out)
const char*	**family** ( ) const
const char*	**type** ( ) const

Table 7.17: GFront Interface

these segments accordingly to try and produce a closed boundary; if the chain fails to close, the front is marked *invalid* according the *Thing's* invalid flag. This prevents any dependent meshes from trying to mesh an open boundary.

The reuse of the *GDSegmentChain* object, in addition to it being the sole member of its family, makes the front a very straight forward object in terms of its class hierarchy. Table 7.17 shows only a public interface, there being no generic interface with concrete representations, with it being the first object, so far, to implement all the required communicative and interactive functions of the *Thing* and *GeometricThing* in one object.

7.8 Meshes

Meshes are defined using either discrete segments or fronts, the spacing of nodes along either of these objects dictating the density of the mesh. Previous geometric objects have used separate representations due to their reusability, the ability

Protected Attributes	
NodeReferenceBuilder*	_nodeReferenceBuilder
ElementReferenceBuilder*	_elementReferenceBuilder
NodeArray	_boundaryNodes
NodeArray2	_holeNodes
NodeArray	_meshNodes
ElementArray	_elements
XPoint	_centre
bool	_centre_cached
XBox	_boundingBox
bool	_boundingBox_cached

Table 7.18: GMesh Representation

to encapsulate data with the mathematical operators, define higher level objects in terms of these representations, and to interact with the display using these representations. Therefore a parallel system can be seen in some objects implementing a geometric point interface alongside a node reference interface. Meshes exist on a node, and element, representation; only the outer boundary can be described using the geometric point interface, facilitating the detection of proximity to other objects. Like the discrete segment inherits nodes from defining points, creating intermediate nodes for itself, the mesh inherits its boundary nodes from the discrete segment or front, creating new nodes describing the mesh it creates. As can be seen from the mesh's representation, table 7.18, node references are maintained within three groups. Boundary nodes define the outermost boundary of the mesh, its perimeter, whilst hole nodes are held in a two dimensional array, an array of boundaries that allows any number of holes boundaries to be defined. Mesh nodes are again held separately. This segregation is maintained throughout the objects forming the final stages of the design. Boundary forming nodes are carefully managed throughout the various building block objects in order to prevent duplicate nodes from existing at the same point in space. This might be a problem when outputting a node and element representation of the machine where different nodes have the same coordinates and are referenced by different elements. Components pose a problem in this scheme, explained later in their section of this chapter, the result being that intersecting components are checked for overlaying nodes on boundaries. Identification of boundary nodes simplifies this check.

7.8.1 Concrete Representations

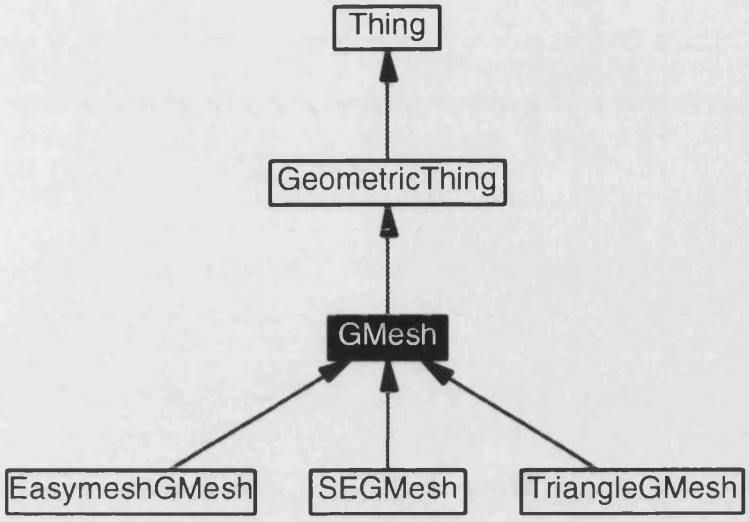


Figure 7.34: Inheritance Diagram For GMesh

The inheritance diagram of figure 7.34 shows the three concrete mesh representations. Meshes generators are complex in their design, therefore one type has been written and the other two make use of third party software.

Super Element Mesh

A Super Element splits a domain into smaller, more manageable, areas, allowing a simpler mesh generation to be used. This type deals with triangular and quadrilateral areas, sides to these areas are supplied in the form of discrete segments. Areas bounded by a greater number of discrete segments can be meshed by consolidating sequential segments using the *CGDSegment* chain object explained earlier. The interface simplifies the object by using the first discrete segment supplied as the seed for the meshing. The mesh generator propagates a wave from this face towards the opposite face, for a quadrilateral, or point, for a triangle. This imposes the restriction that both discrete segments either side of the propagating face must have the same number of nodes. This type does however produce very structured meshes using triangular and quadrilateral elements. Figure 7.35 shows the meshed version of figure 7.31 using five super elements.

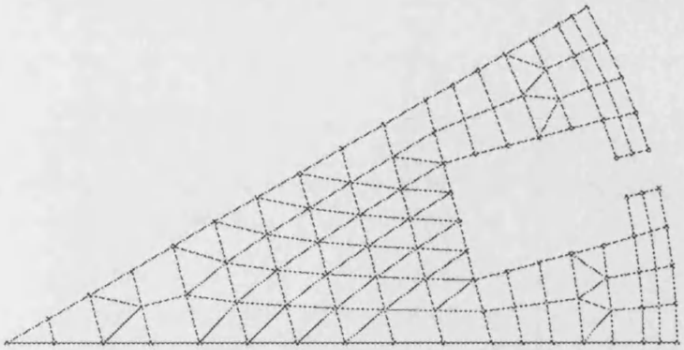


Figure 7.35: Super Element Mesh Using Five Sub Domains

Easymesh

A public domain mesh generator called Easymesh[1], using the Delauney Voronoi method, has been interface into a concrete type. The source code for this program is freely available and was initially integrated into the program. It was found that some geometries caused the mesh generator to fail and thus the whole program terminated; subsequently the two programs now communicate through the node and element file formats Easymesh understands, Easymesh being launched as an external program. In addition to the gained stability, the infrastructure for communication with external programs has been built that has allowed another mesh generator, Triangle[2], to be used. This is useful in terms of the additional mesh generation programs that could be interfaced in order to attain the quality of mesh wanted.

Easymesh handles complex geometries containing holes, an interface of fronts is used to describe the mesh boundary and its enclosed holes. Using the geometry of figure 7.32, we can see the results, in figure 7.36, produced by Easymesh when the area is meshed without the aid of further subdivision into smaller domains. Figure 7.37 then introduces a hole into this boundary.

Triangle

Another use of third party software, Triangle[2] is much faster than Easymesh and it gives greater control over the resulting mesh. Both the minimum an-

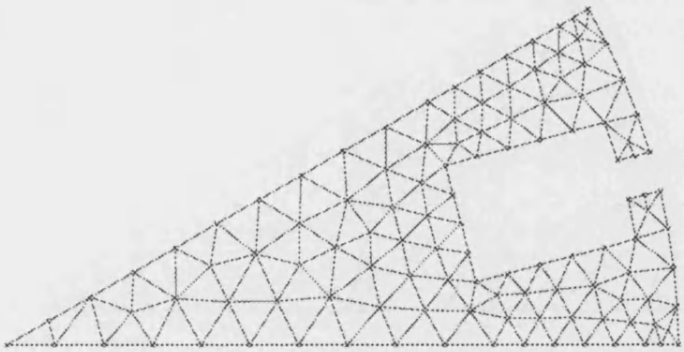


Figure 7.36: Easyesh Based Mesh Using One Domain

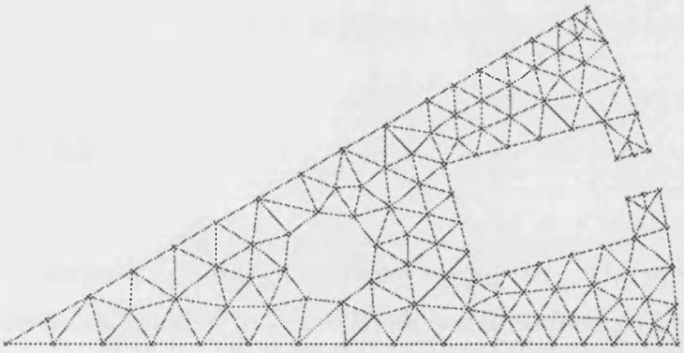


Figure 7.37: Easyesh Based Mesh Using One Domain With Hole

gle size within elements and the maximum area for elements can be adjusted. Figures 7.38 and 7.39 show the meshing of geometries corresponding to those produced by Easymesh in figures 7.36 and 7.37 respectively.

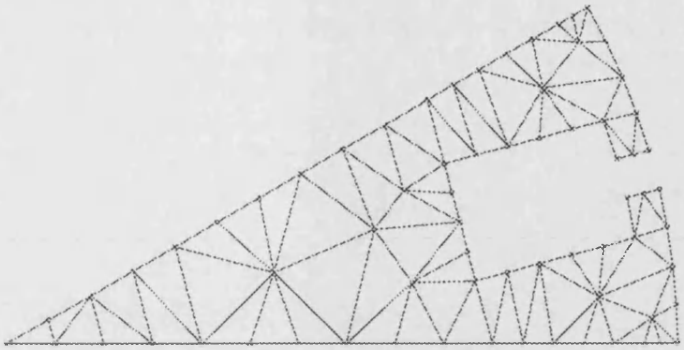


Figure 7.38: Triangle Based Mesh Using One Domain

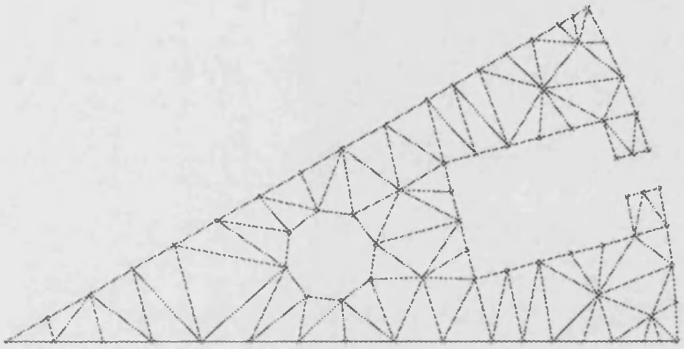


Figure 7.39: Triangle Based Mesh Using One Domain With Hole

7.9 Regions

Regions collate meshes according to material properties. The structure of node and element references founded in the *GMesh* is maintained so that components can easily identify boundary nodes. This object, like the front, has only the concrete representation. It takes a material property in the form of a *GMaterial*, whose representation simply composes of an integer material identifier. This a generic object, simply so derived types can set the property identifier for particular materials allowing the material to be entered descriptively with the likes of "air" or "copper". The other arguments supplied to the region are one or more arbitrarily ordered meshes that take this material property.

7.10 Components and Mapping

The final stage in the machine's development collates all regions into a component. If the symmetry of the machine can be exploited, only this section needs designing to produce the template component. This section can be copied with a mapping policy governing the space the new component is copied to. The *GComponent* object provides the interface for the component, its representation consists of the same node and element arrays used by the *GMesh* and *GRegion*. A *RGComponent* concrete object takes an arbitrary list of regions and places their node and element references into the representation. A mapping *CMGComponent* reads the *GSegment* interface, translating the nodes to produce a new set according to the a supplied mapping. The mapped component stores these new nodes, and corresponding elements, in its representation, so another mapped component could read this object instead of the original region based component. To build a complete machine from a single slot design, a rotational mapping can be used to map the original component onto a new component adjacent to the original; this would use a rotational mapping, the angle parameterised according to the number of desired slots. The machine can be sequentially constructed by taking the last mapping object to translate to the next, utilising just one type of rotational mapping for construction of the entire machine.

Three mapping objects exist, figure 7.40 shows the flipping of a component about a line of symmetry. Figure 7.41 shows the same component rotated about a point, finally figure 7.42 shows a method allowing any kind of translation; two segments can be used such that the mapping is described by translating the *seed* segment onto the *target*. This allows scaling, rotation and translation along any axis.

To complete a section of a machine, all that remains is a component mapping for the ongoing slot example used to illustrate the various building block objects. Figure 7.43 shows the final slot component to be used. The outer, tooth, section is the main area of interest, so this has been meshed using four super element meshes. This has produced some nice quadrilateral elements. The rest of the slot, covered by mesh $m5$, has used the Triangle meshing scheme to allow incorporation of a hole into this boundary. The slot angle has been parameterised, we now make this a function of the number of slots we desire:

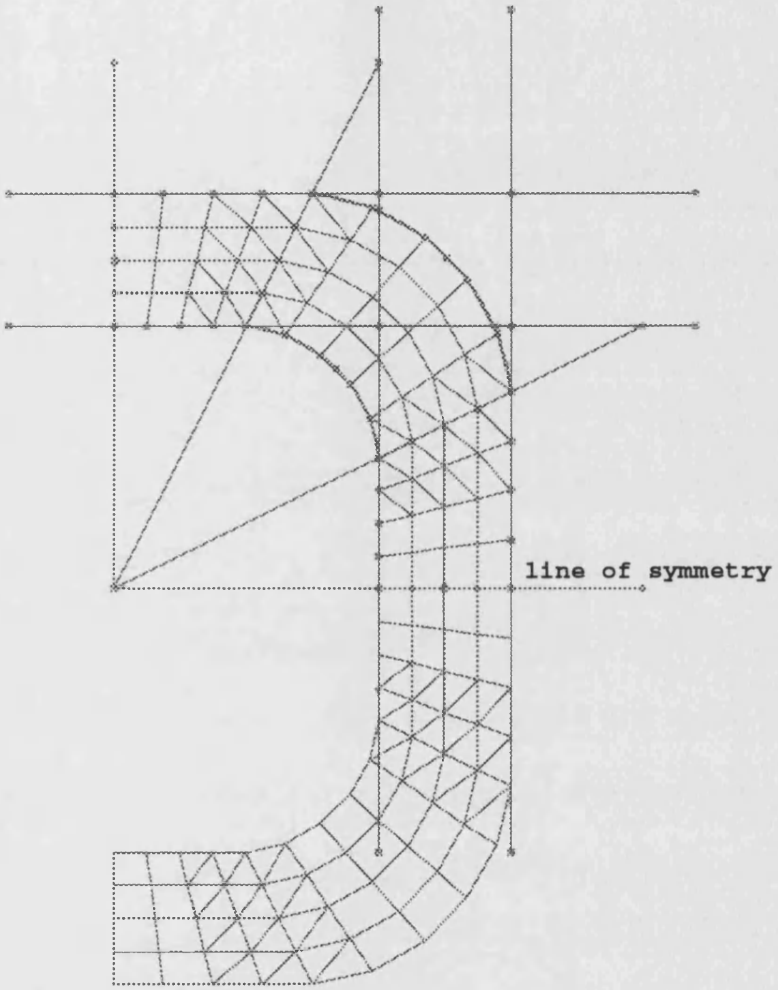


Figure 7.40: Flipping Components About A Segment

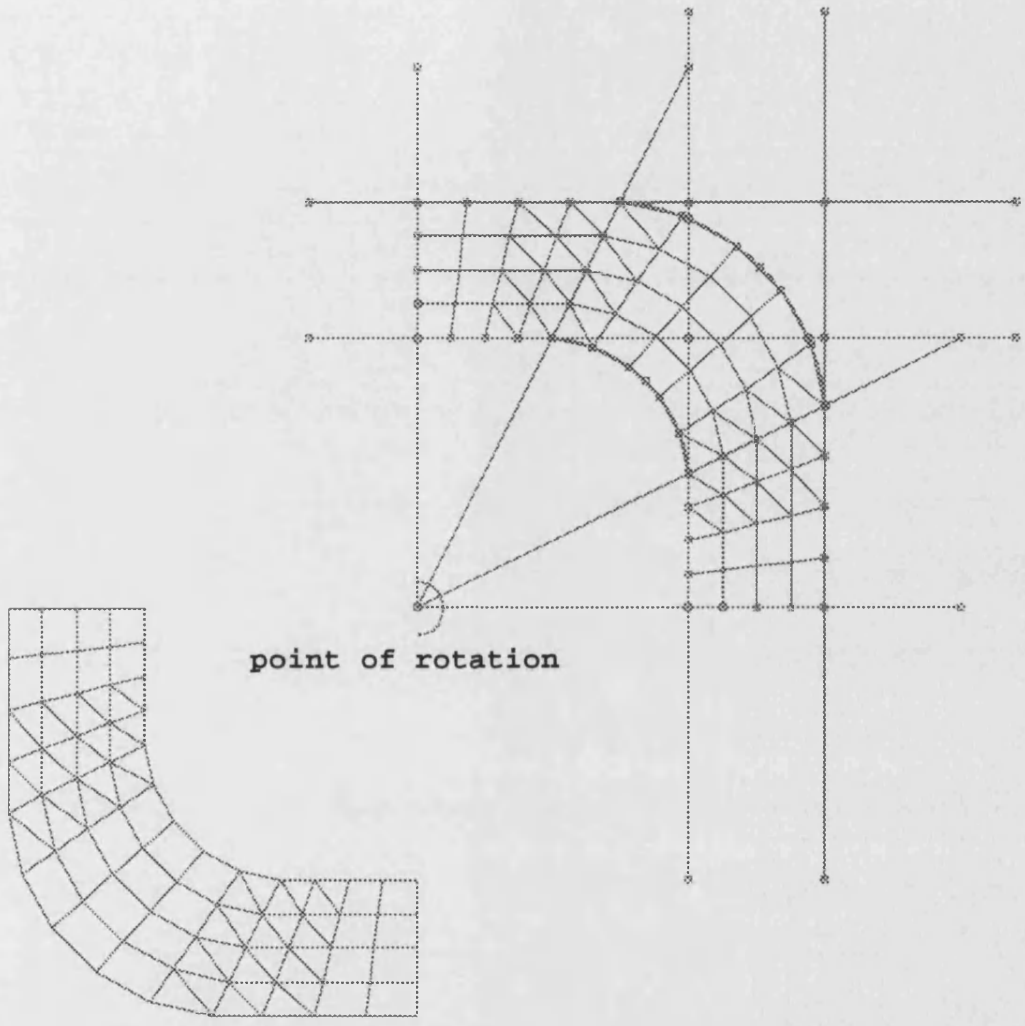


Figure 7.41: Rotating Components About A Point

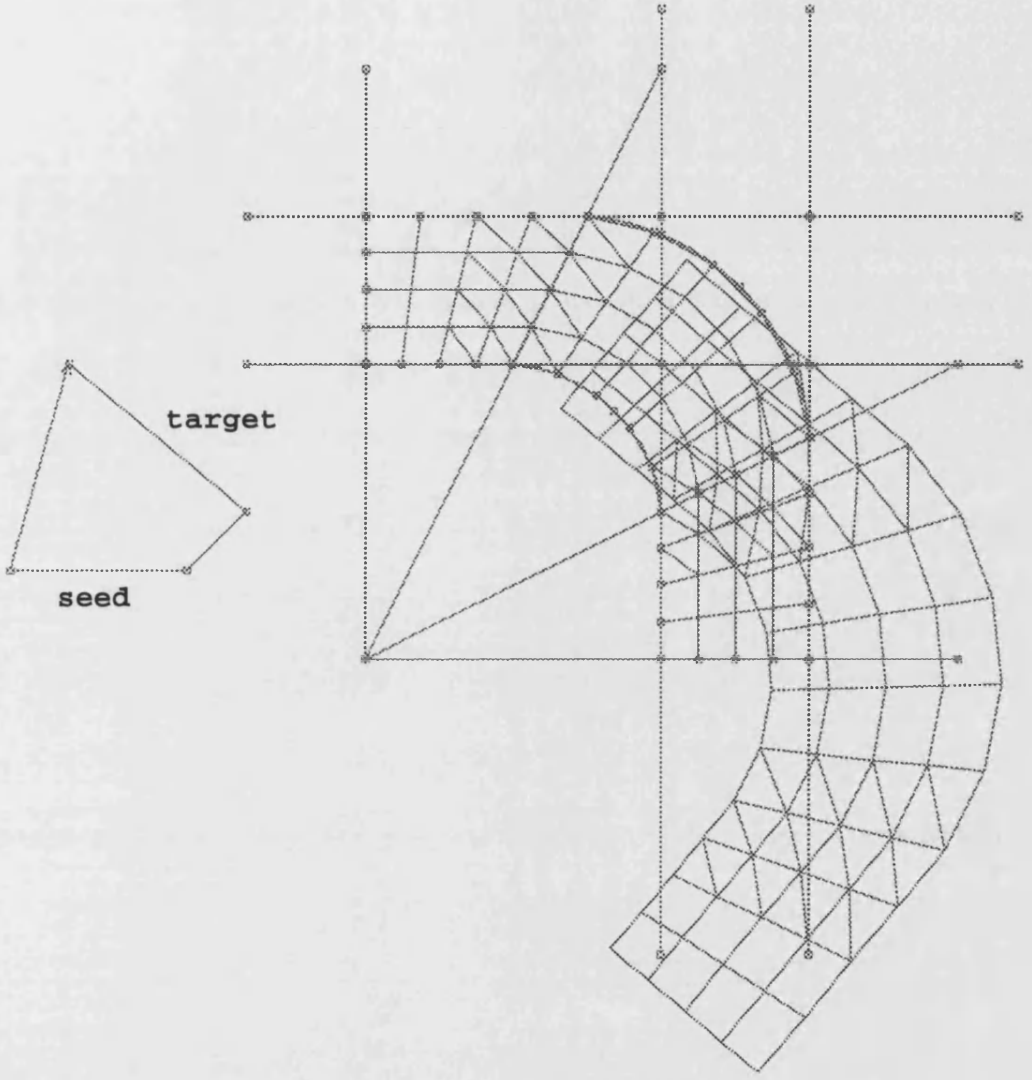


Figure 7.42: Translating Components Using Two Segments

```
slots=12
slot_angle=360/slots
```

Next we define our mapping policy, this rotates the slot about a centre point in the z-axis:

```
mapping=rotate(centre, 0, 0, slot_angle)
```

Finally, the component, named as such, is mapped for all the slots using a command to show the result of figure 7.44:

```
cs component mapping slot_angle
```

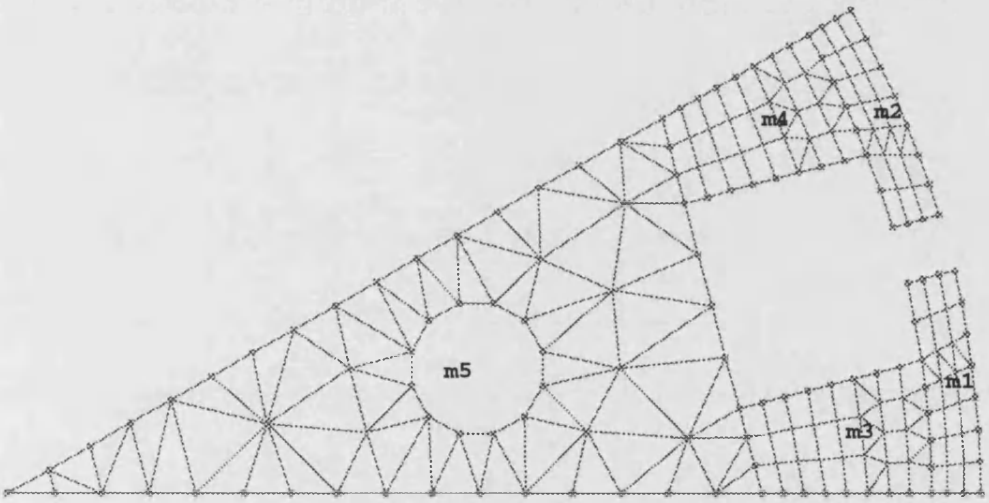


Figure 7.43: Final Slot Configuration

Mapped components pose a problem in the scheme of node references. Throughout the design of the machine, objects have depended upon and overlaid one another, from foundation points to components, in a layered design. Node references have been inherited from one object to another so that overlaying objects don't produce nodes on top of existing nodes. If two nodes overlaid one another on the boundary of two meshes, the elements of one mesh would reference one node and vice versa. The resulting node and element representation of the machine could prove confusing to a solver. Components break this scheme. A mapped

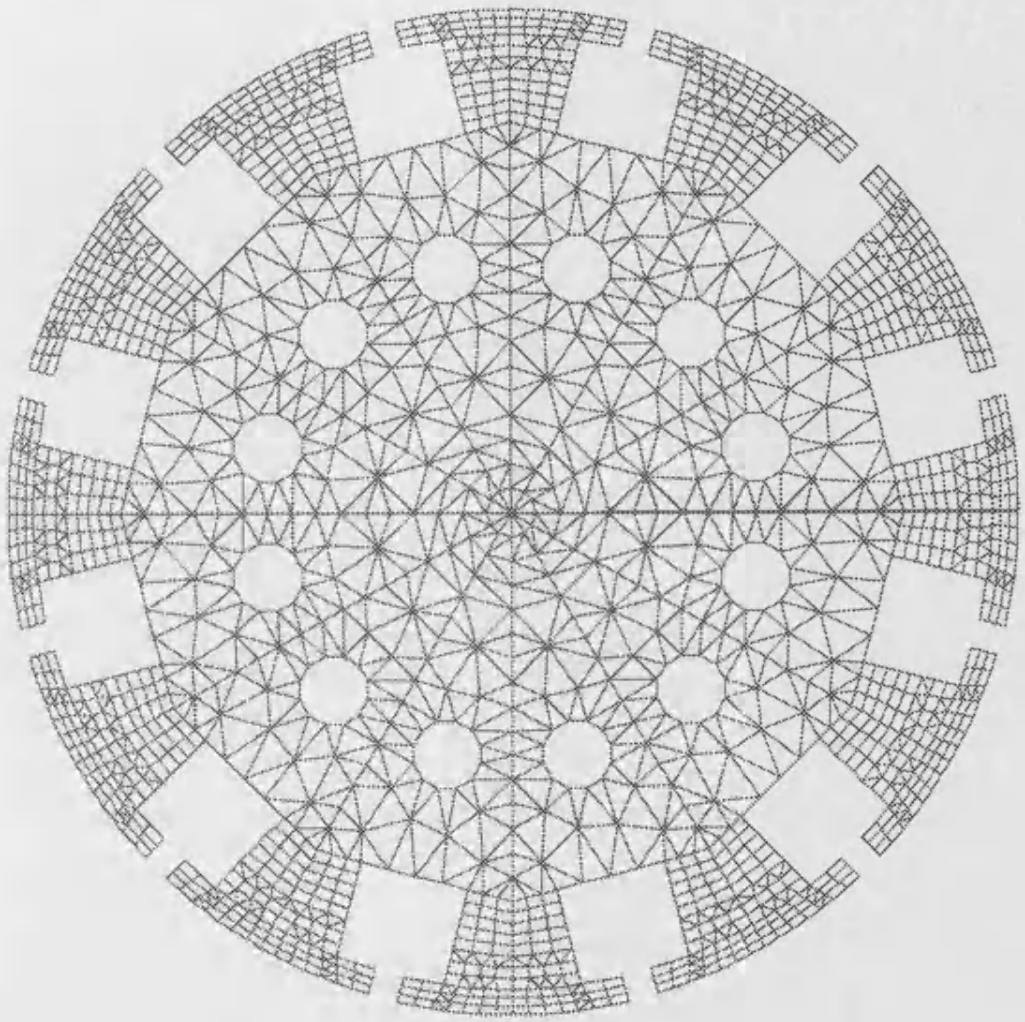


Figure 7.44: Slot Mapped To Produce A Twelve Slot Section Of An Electrical Machine

component knows of only one other component. The mapping may be such that the translated component doesn't ever share a boundary with its master. It could at best ensure that it didn't create new nodes along a shared boundary. However, with a final mapping component fulfilling the symmetry of the machine, closing the gap between two components, this component has no knowledge of component on the opposite side to its parent. Figure 7.45 illustrates this, component *c4* closes the symmetry by bridging components *c1* and *c3*; this component knows of either *c3* or *c1* but not of both. The situation worsens if *c4* depends upon *c2*.

To remedy this problem, we utilise the post-processing of the specialised builder detailed in chapter 6. Already intersecting segments affected by parameter changes, the builder extends this to components by using the *boundingBox* of the *GeomericThing* interface to discover component intersections. Each component intersection is added to a list of intersections held within the respective components. The builder maintains these lists through the duration of the design. Only when the node and element information is output are these intersections examined; then the boundary nodes, which have been maintained separately through the object node inheritance, are examined for duplications.

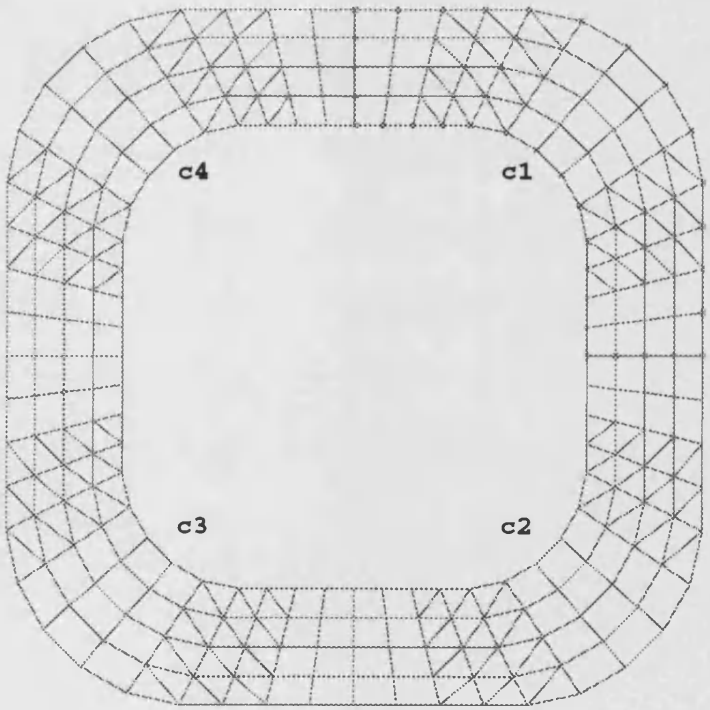


Figure 7.45: Sharing of Component Boundaries

Chapter 8

Conclusions

The objectives of this work have been met.

The electrical machine has been broken down into separate objects using a top-down design that starts with the *component* as the largest of these objects. The component exploits the symmetry of the electrical machine so that geometric mapping of this object constructs the entire machine, releasing the programmer from the repetitive process of entering the entire geometry. Region property mapping then handles the changes in materials throughout the machine, needed to define windings with different phase and polarity.

The component is built using a bottom-up design, starting with *values* that parameterise any aspect of the electrical machine. The geometry is then constructed using the interaction of *points* and *segments*. Line, arc, and circle segments anchor to points and in turn intersect to produce intersection points that provide further anchorage for more segments. This process is used to construct the outline of the component and partition it internally into a number of non-overlapping tiles. Nodes are placed around the outline of the component and along the internal boundaries between the tiles using *discrete segments*. The tiles are then meshed and these *mesh tiles* are grouped into *regions* which group into the final component.

Multiple components can be used within the design of the electrical machine, for example an induction motor can be built from two components, one representing

the slot of the stator and one representing the slot of the rotor. Components are then reuseable and can be stored in libraries for importing into future designs. Each of the separate objects, points, segments, and so forth, is stored within a dependency tree that can be manipulated and edited to make easy changes to components that will ultimately propagate through the entire machine.

The object oriented approach has shown to be very useful in implementing this framework.

Each family of object has been implemented using an abstract, *generic* interface. Each object within that family, the *concrete* objects as seen and used in the design, hides behind this interface. Thus a point can be defined in terms of cartesian and polar coordinates and it can be defined by the intersection of two lines. These different types of point are the concrete types and vary considerably in how they are defined. The abstract interface of the point hides this. Any object that uses a point sees the abstract interface which will provide the point's location, they do not see how that location is derived. This abstraction allows us to attach a line segment to any point irrespective of its concrete type.

The functionality of the objects used in designs can be easily extended as a result of this. If we wish to provide another means of defining a point we simply ensure that it adheres to the abstract interface. As only the abstract interface is seen throughout the rest of the framework we can be sure that this addition will not break any of the existing framework. The designer has the freedom to define geometries in the most convenient form whilst from a programming perspective we needn't check that each new object is capable of interacting with every other object.

Abstract interfaces are neither restrictive in terms of the additions we can make to these families of objects. The segment family defines types of lines, arcs, and circles, that are described in terms of a number of points that lie along their path. Not only can we add to the methods in which lines and arcs can be defined, using additional concrete objects for these types, but we can also add a completely new type of segment such as a spline.

As the basis for all families of object we use a class called the *Thing*. All the families of derive from this Thing and thus can be described in terms of it. The

Thing forms the basis for the dependency tree used to store the relationship between objects once they are entered into the design. When a line segment is constructed in terms of two points it is given a name and constructed in terms of the names of the two points. This relationship is stored in the dependency tree and allows us to efficiently filter changes down the dependency tree that only affect the objects that are dependent on the object being changed.

When changing an object we pluck it from the dependency tree and replace it with one newly defined by the designer, perhaps in terms of other objects. The old object is stored and its place in the dependency tree restored if we choose to undo this change. The use of the Thing ensures this mechanism remains independent of the actual objects being manipulated, knowledge of their specific type is irrelevant and so a generic undo/redo mechanism is provided. The designer is free to take a line segment defined by two points and replace it with a line segment that lies tangential to an arc. The line segment can even be replaced with an arc segment. When components are reused in new designs this flexible interchange of objects allows easy modification of the component towards the new design and any undesired change can be undone and subsequently redone. As a programmer we benefit because we don't have to implement an undo mechanism in every object we add to the framework. Thus we don't risk breaking the framework because additions weren't implemented correctly.

Naming objects gives us a descriptive language for representing the electrical machine which has a one to one mapping on to the dependency tree. This language allows us to make design changes without the need for graphical tools. We can visualise the dependencies between objects and clearly see what constraints we have. With the bottom-up design we always have values at the base of the dependency tree, thus every aspect of the machine is parameterised and can be manipulated at the language level or using design tools.

The ease of making changes the the parameterised design, without human interaction, lends itself to batch processing. A value, such as the slot tooth width, can be varied in a linear manner and the modified mesh generated and solved to see which width yields the best results.

With the additional ability to interpret data from the finite element solver we would have the option of creating an optimisation loop that would streamline

this process, please see the chapter on further work.

Chapter 9

Future Work

This PhD has effectively become a single program which the author would like to see reach maturity. Of the existing program code there is little that needs to be changed. The author has confidently used this program for extensive periods of time without incident throughout the writing of this thesis.

9.1 Graphical Interface

At one stage the program had a very useable graphical interface, effort had been put into this in order to demonstrate reusability of the object oriented structure with application to a graphical user interface. The building blocks of the electrical machine became graphical widgets, like menu bars and buttons, the interaction of these was governed by the dependency tree such that resizing the display filtered dimensions down the dependency tree redimensioning and redrawing widgets.

The interface has since suffered due to the impact of program changes made to improve the electrical machine design. More effort has been put into the necessary changes than to the maintenance of the interface, which has been kept useable. The interface is now the main obstacle that prevents this program from being used in a production environment. Usability of the program would also benefit from more interaction with the designer. For example the ability to point at discrete segments and interactively change their node density would improve the

mesh refinement process. Such tools need a solid foundation to build on, which doesn't exist.

9.2 Post-processing

Post-processing of the finite element model would allow this program to become a complete solution for the design of electrical machines. This depends greatly on improvements to the user interface in order to represent the results visually. However the ability to read in the results of the finite element analysis would also allow a feedback loop to be put into place from which we could optimise the electrical machine.

It is currently possible to batch process refinements to the mesh because the same instructions understood during interactive use can also be deposited in files and run automatically. This allows an element of remote control to the program, however this not an ideal solution. Ideally the program would submit its mesh for analysis and when ready read the results back in for graphical representation. The program would allow specified measurements to be taken anywhere within the model, feeding these back into an iterative optimisation loop that adjusted a parameterised mesh before writing it out and repeating this process. The designer would create the optimisation loop, designating what was to be measured and what parameters were to be correspondingly varied. They would define the relationship between the measurement and the parameters, specifying limits and the degree of variation. The process would complete when the specified condition was met.

Appendix A

An Object Oriented Approach to Parameterized Electrical Machine Design

Entered into the IEEE Transactions On Magnetics, Vol. 36, No.4, July 2000.

An Object Oriented Approach to Parameterized Electrical Machine Design

M.B. Norton P.J. Leonard
University of Bath, Claverton Down, BATH BA2 7AY, UK

***Abstract*—This paper details the structure and advantages of an object oriented, parameterized preprocessor for use in electrical machine design. Detailed are generic objects that form the model base and interface objects that define the design structure. Abstractions within the design are explained detailing additional benefits of the object oriented methods.**

Introduction

Object oriented methods are now demonstrating their benefits to the programmer in the design of reusable, reliable and maintainable programs for electrical machine design[1]. There is no unique mapping from the problem domain to the class structure, in this paper we present our scheme which we believe has significant advantages compared to previous schemes.

Our approach aims to fulfill the needs of both the experienced and less experienced designer of an electrical machine. The user is guided through a scripted design process, answering initial questions detailing machine aspects until a template is provided which provides the canvas for the design. Experienced users can create and modify these templates to satisfy customisation or prepare environments for less experienced users. Ultimately this template takes parameterised plug in components of a machine, either purposely designed or re-used from earlier sessions, and fits them into the global machine. Our approach to designing these components is similar to that of a Computer Aided Design package. We use the object oriented approach to build up components in layers from different object building blocks. The fundamental building block is a value, the family of value objects allow parameterisation of machine slot numbers, dimensions and electrical characteristics. Variation of the values allows different scenarios to be tested. Building on top of the values, segments provide the lines and arcs which form the physical shape. Objects of a family are interchangeable by virtue of their common interface, allowing the designer to iterate through different designs. Ultimately mesh and region objects produce re-usable segments that may be symmetrically exploited within the machine and exported to libraries for future re-use. All objects are communicative; they are the nodes of a dependency tree which relate between object families using their generic interface. Changes in any object will filter through the dependency tree updating the total model. External manipulation of the dependencies is possible to allow variations in the design through its parameterisation; with feedback this allows iterative analytical solutions to be parsed in order to automatically find the optimised parameters.

Manuscript received October 25,1999

Object dependency

A dependency object forms the foundation of all the objects seen by the user. When an object is constructed it is supplied with a named list of objects. The constructed object reads its constraining data through the interfaces of these objects. If their state changes, they notify this object and other dependent objects of their change; dependent objects are then able to update their state by reading the new data. An object family such as segments is represented by lines parameterised from zero to one. A circle segment could be described using three points or a centre and radius. Whichever method is chosen, the resultant circle can always be parameterised; this is the generic interface. The two circle interfaces mentioned are the concrete representations. All object families inherit the dependency object and build on top of it the interface to that family; this is then inherited by concrete objects, figure 1 details this inheritance.

Once a family interface is defined, all a concrete object needs to do is convert its definition into the generic representation. The tools the designer has to work with are now easily extendible through the addition of concrete objects.

Figure 1 shows part of the class hierarchy illustrating the dependency between concrete classes and the generic interface, defined in terms of an abstract base class. We have only shown some of the relationships to avoid clutter.

The user's benefit of this aspect is in the ability to pull one type of object out of the dependency tree so as to replace it with another object of the same type. Now the user can modify a design, as demonstrated by figure 2, by interchanging objects. The design process is not restrained by previous actions, the user can iteratively experiment by interchanging concrete types at any point, even after subsequent objects have been made dependent

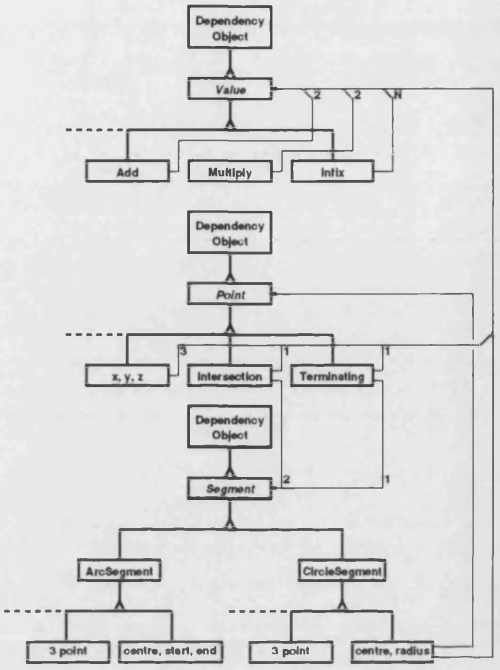


Fig. 1. Subset of class hierarchy showing typical relationships between concrete and generic objects

opon them.

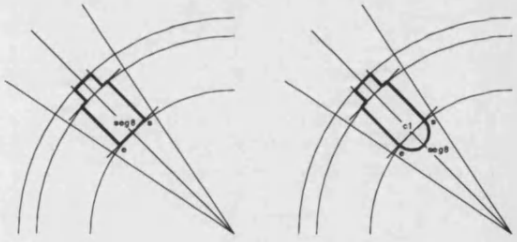


Fig. 2. Generic object interchange

Object Validation

Figure 2 shows the rudiments of a slot geometry constructed from a set of arc and line segments. The segment family's function is to provide some kind of parameterised line. Segments may then be placed to intersect, points are positioned at the intersection and regions can be defined by walls created along segments between two points. Intersection points are automatically created in the build process, they take two segments and update their position according to the segment's intersection. Should the segments no longer intersect, the point, and any of its descendents, are marked invalid. Invalid objects are alerted to the user, should they wish to delete them, otherwise they lie inactive until the model changes such that they can be revalidated appropriately.

Providing Constraints

Sometimes a segment's ends are constrained by being attached to defining points, other segments defined differently may create these points so they exist. The points then facilitate connection to other objects. Differences in these types lie in the order of dependency. They differ significantly in their constraining behavior. The dependency tree governing the model filters changes down the tree so behavior is dictated by dependency. Rather than needing explicit constraints, the dependency relationship implicitly defines these; a straight line dependent on two points must vary it's length according to the distance between points, whereas a line defined by a starting point, angle and length will create end points which must move as the line's length changes. This constraining effect simplifies the desired parameterization response within a model and it's effects are immediate. It avoids the cost of iterating associated with approaches that effect a change, then iterate over the simultaneous equations, coupling all objects until a solution is reached[3].

Constructing Through Prototypes

The construction of objects is simplified through the use of prototypes[2]. For every concrete object a prototype exists. This prototype knows of the generic object types its counterpart requires and understands textual expressions detailing its construction. Given the expression:

```
p = point(x,y,z),
```

the given variables x, y, and z are checked by the prototype to ensure they are of the correct type, additional checks are also possible. In validation the prototype consults the model, should the variable not exist an object builder will try to construct it. This allows the nesting of further expressions within the body of the main expression:

```
p = point(x,5,z); l = line(point(x,10,z), p)
```

The Nice Side-Effects

The prototype simply works through the expression resolving variables, constructing where necessary. Constructed variables are given manufactured names unique to the main variable being constructed; should that variable be deleted, then the manufactured variable can be deleted also. Obviously there comes the case where the prototype fails to resolve the necessary variables and the object cannot be constructed. For the case

```
p = point(1,2,z)
```

where z is non-existent, the integer value objects for 1 and 2 will have already been constructed before the prototype fails at z. A history mechanism is required in order to delete these objects on failure if we're to provide a tidy design environment. This has a side effect; the object builder now has a record of the objects created to allow reversal. This provides an undo buffer, where the action had failed it may also have succeeded with this history being stored. Later that history can be recalled, reversed, and the model restored to a previous state.

Taking this a step further, we have a model where communication exists between objects using generic interfaces. Providing the dependency tree structure is maintained, it is possible to extract an object from the dependency tree and replace it with an object of the same interface, i.e. replace an advancing front mesh with a Delaunay mesh. This action is recorded in the history so it can be undone. The designer can now interchange objects of the same family. Walls of meshes may be reshaped with the interchange of segment types, as well as parameterisation. The dependency communication allows the resolution of the mesh to be modified through the values describing its node numbers. Every one of these actions is reversible through the object history. Reversed histories simply get shuffled into another list allowing them to be redone, the user now has the ability to undo and then redo any changes. The changes this implicates are filtered down the dependent objects amending their state. This functionality is common to the dependency object and is inherited by our building block objects; it is generic, no implementation specifics are required; this is unlike other schemes requiring individual methods for concrete classes[4]. The history understands simultaneous model changes allowing any sizeable change.

Building a Simple Parameterised Geometry

Using the geometric object types to design the physical aspects of a very basic rotor slot, figure 3, initial values can be constructed. These govern the fundamental aspects of parameterisation, they will be controlled by our global machine. The slot angles, for example, will be set to fit this geometry into the template machine. The following values allow us to demonstrate the building of a simple slot:

```
! centre point, radius and angles for the slot
ctr = point(0, 0, 0)
rad = 20.5; ang1 = 90; ang2 = 90

! some more variables govern slot dimensions
d1 = 10 ! slot depth
d2 = 3  ! slot tooth depth
w1 = 5  ! width between slot teeth
w2 = 10 ! slot width
```

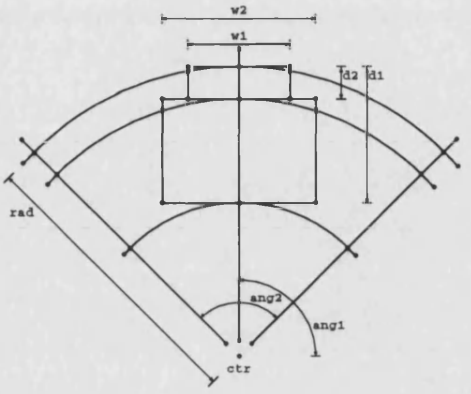


Fig. 3. Build of a basic slot geometry

Now some construction segments are used to create the initial framework:

```
l1 = pvvvline(ctr, 1, rad+1, ang1)
l2 = pvvvline(ctr, 1, rad+1, ang1+ang2/2)
l3 = pvvvline(ctr, 1, rad+1, ang1-ang2/2)
r1 = rad-d1; r2 = rad-d2
a1 = pvvvarc(ctr, r1, ang1-ang2/2+5, ang2+10)
a2 = pvvvarc(ctr, r2, ang1-ang2/2+5, ang1-ang2/2+5)
a3 = pvvvarc(ctr, rad, ang1-ang2/2+5, ang2+10)
l11 = spvvvline(a1, _i1_a1_l1, w2/2, w2/2, 0)
l15 = ppline(_ps_l12, _ps_l11)
l16 = ppline(_pe_l12, _pe_l11)
```

For geometric objects, a point and click interface facilitates the design process by producing the equivalent of the above command lines. For the line:

```
l11 = spvvvline(a1, _i1_a1_l1, w2/2, w2/2, 0),
```

the *_i1_a1_l1* variable denotes the *first* automatically manufactured intersection point between arc *a1* and line *l1*. For the lines:

```
l15 = ppline(_ps_l12, _ps_l11)
l16 = ppline(_pe_l12, _pe_l11)
```

the *_ps_l12* variable denotes the manufactured termination point for the *start* of line segment *l12*, likewise the variable *_pe_l12* denotes the *end* point manufactured for that segment. The next step uses the construction segments and points to define closed areas for meshing. The point and click interface makes this much easier, however it still constructs the same commands to next define the discrete segments. Here, two points are taken which lie on a segment. This sub-segment is given a policy of discretisation, linear, exponential, user supplied, which it uses to divide that sub-segment into the given number of *n-1* segments, i.e. by defining n points. Ultimately, these are the nodes along the edges of our mesh. The following commands discretise and mesh our slot of figure 3, as shown in figure 4:

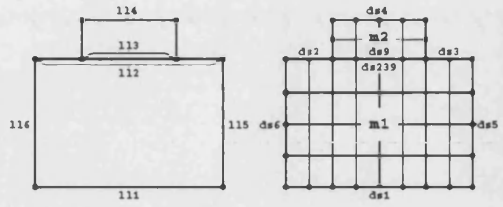


Fig. 4. Build of discrete segments from construction segments

```
! simple control of density through one variable
n=1
! discrete segments for larger area, mesh m1
ds1 = dsegment(l11, _ps_l11, _pe_l11, linear, 9n)
ds5 = dsegment(l15, _ps_l11, _ps_l12, linear, 5n)
ds6 = dsegment(l16, _pe_l11, _pe_l12, linear, 5n)
ds2 = dsegment(l12, _ps_l12, _ps_l13, linear, 3n)
ds3 = dsegment(l12, _pe_l13, _pe_l12, linear, 3n)
ds9 = dsegment(l12, _ps_l13, _pe_l13, linear, 5n)
! combine three segments into one for meshing
ds239 = cdsegment(ds2, ds3, ds9)
! super element mesh
m1 = semesh(ds1, ds5, ds6, ds239)
```

In figure 4 a super element mesh has been used to precisely mesh the area, giving very uniform results. A further layer may be inserted between the discrete segments and mesh; a *front* orders an arbitrarily ordered list of segments to produce a continuous chain of nodes. Meshes may then take fronts as arguments allowing holes within meshes.

```
! create one front for the slot
fr1 = front(ds1, ds2, ds3, ds4, ds5, ds6, ds7, ds8)
! mesh the front using a delauney voronoi mesh
! this handles holes,
! mesh = dvmesh(boundary, hole1, hole2, ...)
m1 = dvmesh(fr1)
! try out an advancing front mesh instead
m1 = afmesh(fr1)
```

The Final Machine

All that remains is to group meshes together with a material to form a region; regions can also be grouped together to form components, acting as containers. All these object types can be mapped onto a new object of the same type. This simply allows duplication of an object, the duplicate mirroring the original so that any changes to the original affect it's mapped duplicates. The component object, when mapped, will allow us to exploit any symmetry within the machine by using an appropriate mapping, such as rotation or mirroring along a line. With our generic object interfaces, mapped objects can further be mapped themselves. Should we need to customise a mapped object, because of some peculiarity, we then exploit the copy on write aspect of these objects to make the copy into its own entity that can be modified separately. However, a more elegant solution in line with our machine template allows us to re-use our work in a future project. So far the slot has been built as a separate entity from the machine. At this point we can export, as we could at any earlier stage, the component so it can be inserted into any future project. The exported library component contains the necessary information to reconstruct our geometry, along with a list of the key variables that parameterise the geometry. For our example, this would include the angles, centre, radius and maybe the slot dimensions. The export process allows descriptions of these variables to facilitate the import at a later stage. This allows libraries of useful geometries to be constructed by more experienced designers for the less experienced to re-use later. The motor templating scheme can automatically link in these defining variables to a global set; geometries imported into the template become globally controlled this way. Imported components, if modified, can be re-exported to further enhance the library.

Conclusion

Use of abstract mechanisms allows object interactive methods to be independent of implementation, interfaces, file transfer, model construction and modification need no knowledge of specific implementations; this allows simple extension of object types to facilitate the design process and accommodate the needs of the most experienced of users. A dependency structure allows easy understanding of model constraints, object interchange, undo and redo allow easy restructure of the model. Parameterization facilitates this leading to automatic model optimization, further aiding the design process. Templates for machine design allow automation and customisation of machine design to suit the level of the particular user, exploiting the export and import of library components to re-use designs; these being easily modified through object replacement and the parameterisation to adapt a re-used design to a new scenario.

References

[1] Renato C.Mesquita, Renato P.Souza, Túlio Pinheiro, Ana L.C.C.Magalhães, An Object-Oriented Platform for Teaching Finite Element Pre-Processor Programming and Design Techniques, IEEE Transactions on Magnetics, Vol. 34, No. 5, pp. 3407-3410, September 1998.

[2] E.Gamma, R.Helm, R.Johnson, J.Vlissades, "Design Patterns:" elements of reusable object orientated software, Reading: Addison-Wesley, 1994.

[3] C.F.Parker, J.K.Sykulski, S.C.Taylor, C.S.Biddlecombe, Parametric Environment for EM Computer Aided Design, IEEE Transactions on Magnetics, Vol. 32, No. 3, pp. 1433-1437, May 1996.

[4] Ana Liddy Cenni de Castro Magalhães and Renato Cardoso Mesquita, Requirements for a Solid Modeler Coupled to Finite-Element Mesh Generators, IEEE Transactions on Magnetics, Vol. 34, No. 5, pp. 3447-3450, September 1998.

References

[1] B. Niceno, "Easymesh: A two-dimensional quality mesh generator." http://www-dinma.univ.trieste.it/~nirftc/research/easymesh.

[2] J. R. Shewchuk, "Triangle: A two-dimensional quality mesh generator and delaunay triangulator." http://www.cs.cmu.edu/~quake/triangle.html.

[3] S. T. S. P. Electronics and E. D. Consortium, *MEGA*. University Of Glasgow, UK. http://www.speedlab.co.uk/index.html.

[4] T. J. R. I. E. T. Division, *JMAG-Studio*. The Japan Research Institute. http://www.jri.co.jp/pro-eng/jmag/e/jmg/.

[5] H. Assadipour, *Learning AutoCAD in 20 Projects*. St. Paul, MN: West Publishing Company, 1994. ISBN 0-314-02837-4.

[6] T. J. Tautges, "The common geometry module (cgm): A generic, extensible geometry interface," *Proceedings, 9th International Meshing Roundtable*, pp. 337–348, October 2000.

[7] M. J. P. R. Sahu and W. H. Gerstle, "An object-oriented virtual geometry interface," *Proceedings, 6th International Meshing Roundtable*, pp. 67–82, October 1997.

[8] P.K.Vong, H.C.Lai, and D.Roger, "Optimization of electromagnetic devices using parameterized templates," *IEEE Transactions on Magnetics*, vol. 37, pp. 3538–3541, September 2001.

[9] M. C. Bastarrica and N. Hitschfeld-Kahler, "An evolvable meshing tool through a flexible object-oriented design," *Proceedings, 13th International Meshing Roundtable*, pp. 203–212, September 2004.

[10] B. R. Simpson, "Isolating geometry in mesh programming," *Proceedings, 8th International Meshing Roundtable*, pp. 45–54, October 1999.

[11] A. Telea, "An object oriented fem system." http://www.win.tue.nl/ alext/ALEX/PAPERS/papers.html.

[12] A. L. C. de Castro Magalhães and R. C. Mesquita, "Requirements for a solid modeler coupled to finite-element mesh generators," *IEEE Transactions on Magnetics*, vol. 34, pp. 3447–3450, September 1998.

[13] C.F.Parker, J.K.Sykulski, S.C.Taylor, and C.S.Biddlecombe, "Parametric environment for em computer aided design," *IEEE Transactions on Magnetics*, vol. 32, no. 3, pp. 1433–1437, 1996.

[14] D.Roger, H.C.Lai, and P.J.Leonard, "Coupled elements for problems involving movement," *IEEE Transactions on Magnetics*, vol. 26, pp. 548–550, March 1990.

[15] A. E. R. Centre, *MEGA*. University Of Bath, UK. http://www.bath.ac.uk/Centres/AERC/mega.html.

[16] P. A. . C. Liu, *DNS and BIND*. Sebastopol, CA: O'Reilly & Associates, Inc, 3 ed., September 1998. ISBN 1-56592-512-2.

[17] A. Rudd, *Mastering C*. John Wiley & Sons, December 1993. ISBN 0-471-60820-3.

[18] B. Stroustrup, *The C++ Programming Language*. Reading, Mass.: Addison-Wesley Publishing Company, 2 ed., 1991. ISBN 0-201-53992-6.

[19] B. Stroustrup, *The C++ Programming Language*. Reading, Mass.: Addison-Wesley Publishing Company, 3 ed., September 1997. ISBN 0-201-88954-4.

[20] D. van Heesch, "Doxygen." http://www.stack.nl/~dimitri/doxygen/.

[21] E.Gamma, R.Helm, R.Johnson, and J.Vlissades, *Design Patterns: elements of reusable object orientated software*. Reading, Mass.: Addison-Wesley Publishing Company, July 1994.

[22] A. Bowyer and J. Woodwark, *A programmer's geometry*. Butterworths, 1983.